CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Department of Computer Science
Chair of Computer Architecture

Diploma Thesis

Analysis and Adaption of Graph Mapping Algorithms for Regular Graph Topologies

Sebastian Rinke
sebastian.rinke@cs.tu-chemnitz.de

22th April 2009

Advisor: Dipl.-Inf. Torsten Mehlan
Supervisor: Prof. Dr.-Ing. Wolfgang Rehm

Rinke, Sebastian
Analysis and Adaption of Graph Mapping Algorithms for Regular Graph Topologies
Diploma Thesis, Department of Computer Science
Chemnitz University of Technology, September 2009

Task Formulation

The MPI standard offers facilities for an optimized mapping of processes onto processor elements. This diploma thesis is to investigate different approaches for performing such an optimized mapping on regular network topologies (e.g. grid, torus, hypercube, ...). A prototypical implementation is to be integrated into an existing MPI implementation. Eventually, experimental results are to show the effect of the algorithms found.

Publication

Parts of this work have been submitted under the title *Evaluation of Task Mapping Strategies for Regular Network Topologies* to: *International Conference on Parallel Computing 2009 (ParCo2009), France, Lyon, 1 – 4 September 2009.*

Statutory Declaration

I declare that I have developed and written the enclosed diploma thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The diploma thesis was not used in the same or in a similar version to obtain an academic degree or is being published elsewhere.

Chemnitz, 22th April 2009

Sebastian Rinke

Abstract

The Message Passing Interface (MPI) standard defines virtual topologies that can be applied to systems of cooperating processes. Among issues regarding a more convenient namespace this may be used to optimize the placement of MPI processes in order to reduce communication time. That means, the processes with their main communication paths represent a graph that has to be cost efficiently mapped onto the graph representing the actual communication network. In this context, this work analyses and compares state-of-the-art task mapping strategies with respect to running time and their quality of solutions to the MPI mapping problem. In particular, the focus is on generic strategies that can be used for arbitrary process/network topologies although, here, the topologies of interest are regular ones, where the number of processes is greater than the number of processors in the underlying physical network. Additionally, different measures of mapping quality are discussed and a close correspondence between the most appropriate, the weighted edge cut, and program execution time is shown. In order to investigate how mapping quality affects MPI program execution time, some mapping strategies have been incorporated into Open MPI. Finally, benchmark results prove that optimized process-to-processor mappings can improve program execution time by up to 60%, compared to the default mapping in many MPI implementations (linear mapping). The findings in this work can serve as reference not only for MPI implementors, but also for researchers investigating static process-to-processor mappings, in general.

Theses

I. Optimized process-to-processor mappings can reduce MPI program execution time.

II. The weighted edge cut mapping cost function is suitable for assessing mapping quality in terms of communication time.

III. Open MPI offers facilities for easily implementing a non-trivial MPI topology mechanism.

IV. Recursive graph bipartitioning yields better mapping quality than direct k-way partitioning for the regular graph topologies under consideration.

V. For guiding network topology aware mapping strategies in computing process-to-processor mappings that favour nearest neighbour communication, a communication cost matrix is appropriate.

VI. There is no mapping strategy that is superior to all the others for all possible combinations of process and network topology.

VII. The default linear mapping strategy, which is implemented in many MPI implementations, is able to compute optimal process-to-processor mappings.

VIII. An appropriate combination of process topology and network topology is the basis for optimizing the placement of processes onto processors.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Since the early beginnings of High Performance Computing (HPC), there has been a steady increase in both the number and the size of installations of parallel computers. For instance, in 2004, according to the TOP500 [The] (June 2004)[1], the fastest supercomputer in the world was *Earth Simulator* with 5120 processor cores and 35.86 TFlops. Today, the most recent (Nov. 2008)[2] TOP500's number one is *Roadrunner* which consists of 129600 cores, and has a performance of 1105 TFlops. A broader view yields Figure 1.1 which depicts the development of the number of processor cores in the systems listed in the TOP500 from 1993 to 2008. Generally, it can be seen that the share of systems with a certain processor core count gradually increases to some point after which it decreases and finally disappears. Moreover, while some processor core counts disappear new installations with greater counts are introduced. Thus a pattern can be observed that clearly shows a continuous increase in the number of processor cores in parallel systems over time.

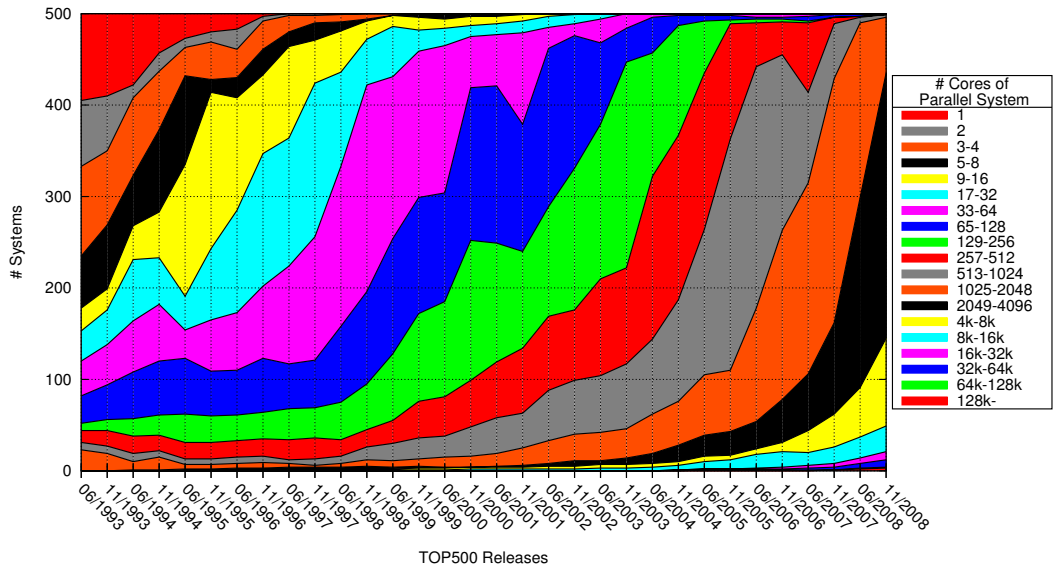


Figure 1.1: TOP500's number of processor cores share over time 1993-2008 ([The])

Regarding this development, the communication between processes and everything related to this, such as the interconnection network, has gained much more attention than in the beginnings of HPC. Meanwhile, compared to the high performance of current processors (Flops), the time required for communication between them has become a serious factor, if not even a bottleneck, for a parallel system's performance.

One possibility to reduce communication time of a parallel program on a given parallel machine is to have a closer look at the communcation pattern between its processes. Often it is the case that each process has a fixed set of communication partners for the greatest part of its execution. Thus the idea is to place heavily communicating processes onto nearby processors in order to reduce the time needed for communication, and thus the total execution time of a parallel application. The term nearby may have

[1] http://www.top500.org/lists/2004/06
[2] http://www.top500.org/lists/2008/11

different meanings which are explained in more detail below. At this point it is sufficient to associate nearby with little communication time.

The Message Passing Interface (MPI) standard [SOHL+98], the de facto standard in message-passing programming, already offers facilities for such an optimized mapping of processes onto processor elements. However, at the time of this writing, only few proprietary implementations of MPI, such as BG/L MPI, MPI/SX, and HP MPI, actually implement optimized mappings. Most of the existing MPI libraries (MPICH[3], LAM/MPI[4], Open MPI[5], etc) simply yield the identical mapping and thus no optimization is performed.

1.2 Scope of this Work

In essence, this work deals with the process mapping facility in MPI and its efficient implementation. Here, the term efficient refers to computing process to processor assignments in time polynomial in the input size, i.e. the number of processes and processors. Additionally, when applied to systems of cooperating processes, such assignments should yield communication times less than those of the default mappings.

Since the mapping problem in terms of MPI can be dealt with in various ways, this section more precisely defines the focus. As already mentioned in the task formulation, the network topologies, i.e. the arrangement of processors in the interconnection network, under consideration are regular ones:

- Fully connected network
- $\{1, 2, 3\}$D grid
- $\{1, 2, 3\}$D torus
- Hypercube
- Binary tree

Similarly, the communication pattern of processes can be described by a graph, the process topology. In practice, often those topologies are regular, as well. Thus the process topologies selected are:

- $\{1, 2, 3\}$D grid
- $\{1, 2, 3\}$D torus
- Hypercube

Those topologies are also known as cartesian topologies in MPI. Hence, the attention is moved to the cartesian topology mapping facility in MPI.

One further aspect is the ratio of the number of processes to the number of processors which the processes are to be mapped onto. As it is common to equip compute nodes with multiple CPUs each (SMP), it has also become common to use multicore CPUs. Examples are *Chemnitzer Hochleistungs-Linux-Cluster (CHiC)*[6] and the top 3 of the current (Nov. 2008) TOP500: *Roadrunner*[7], *Jaguar*[8], and *Pleiades*[9]. Hence, in the following, the number of processes is assumed to be 4 times the number of processors. Note that the terms processor and node are used interchangeably below.

The last important remark concerns the mapping strategies considered. To provide an MPI mapping facility implementation which is generic in the sense that it supports as many process/network topology combinations as possible, it was decided to investigate mapping strategies that don't have restrictions regarding this issue.

1.3 Related Work

Having defined the scope of this work, it is possible to look at related work. Related work, in this case, does not mean to give an overview of different mapping strategies. This is done in a subsection below and is only one aspect of this work. In this context, it would be more appropriate to refer to literature which compares and evaluates state-of-the-art mapping schemes regarding different process and processor topologies, as it is done in this writing. Unfortunately, most of the publications dealing with mapping processes onto processors focus on introducing new approaches rather than relating them to each other.

[3] http://www.mcs.anl.gov/research/projects/mpich2
[4] http://www.lam-mpi.org
[5] http://www.open-mpi.org
[6] http://www.tu-chemnitz.de/chic
[7] http://www.top500.org/system/9707
[8] http://www.top500.org/system/9708
[9] http://www.top500.org/system/9832

For that reason, this section gives an overview of some MPI libraries that already implement an optimized mapping facility.

1.3.1 BlueGene/L MPI

BG/L MPI [AAC+04] is the MPI implementation for the BlueGene/L supercomputer, and is based on the MPICH2 library. BlueGene/L has 2 interconnection networks: 3D torus and tree. The torus is used for MPI point-to-point messages, whereas the tree is designed to perform MPI collective operations. Regarding the MPI process mapping mechanism, the 3D torus is the network topology of interest. That means, process topologies are to be mapped onto a 3D torus or its corresponding sub-topologies. In [YCM06] Hao Yu et al describe techniques for mapping 3D or lower-dimensional grid/torus process topologies onto 3D or lower-dimensional grid/torus network topologies. Their idea is to use graph embedding operations which can be computed in $O(1)$ in parallel. The resulting mapping library has been integrated in BG/L MPI. The drawback of the embedding approach, however, is that it is only applicable to a limited number of process/processor topology combinations.

A different approach was chosen by Bhanot et al. In [BGH+05] they use the optimization technique simulated annealing (SA) coupled with the divide-and-conquer method. This addresses the shortcoming of the embedding technique above. However, the running time of the strategy and the quality of the resulting mapping very much depend on the input (process and processor topology) and the annealing scheme in the SA.

1.3.2 MPI/SX

The MPI/SX library [GRTZ03] is used for message passing on NEC SX systems. The SX-series is equipped with SMP compute nodes where intra-node communication is performed via shared memory. For inter-node communication, those nodes are connected by means of a crossbar switch. In [Trä02, Trä06], J. L. Träff proposes a graph partitioning strategy based on the Kernighan-Lin heuristic for implementing the MPI topology functionality in MPI/SX. The idea is to take an initial mapping of processes onto processors and improve it in a greedy fashion while preserving a global view, i.e. all processes and processors are considered simultaneously, instead of subsets of them only.

1.3.3 HP MPI

A similar approach to that in MPI/SX can be found in HP MPI [HP-07, MPI options p 140]. Here, the underlying physical communication architecture is assumed to be hierarchical. While communication costs differ at different levels in the hierarchy (e.g. shared memory vs. TCP/IP), the costs for all pairs of possible communication partners at the same level are assumed to be equal, i.e. uniform. In [Hat98], T. Hatazaki describes how to recursively apply graph partitioning at each level in the communicaton hierarchy in order to obtain a process-to-processor mapping. The partitioning algorithm used is the Kernighan-Lin heuristic. For performance tests, Hatazaki uses the HP Exemplar X-class architecture which has 3 levels for communication (from fast to slow): hypernode shared memory, global shared memory, TCP/IP network.

1.4 Overview

This work is organized as follows. After the discussion of the MPI standard's facilities for optimizing process-to-processor mappings (Chapter 2), Chapter 3 investigates the process-to-processor mapping problem in more detail. In particular, different cost functions are introduced by means of which mapping quality can be assessed. Furthermore, regarding networks with non-uniform communication cost, approaches for modelling communication cost between pairs of processors are examined. Finally, a formal description of the mapping as dealt with in this work is given. In order to show how mapping quality is affected by the choice of the mapping strategy, Chapter 4 describes the algorithms that are examined in Chapter 5 regarding their potential for improving mapping quality in both networks with uniform and networks with non-uniform communication cost. In connection with this, benchmarks are performed for the process and network topologies described in Section 1.2. In Chapter 6, the integration of some of the algorithms from Chapter 4, in terms of the MPI topology mechanism, into an existing MPI implementation is considered. For that purpose, Open MPI is chosen. Afterwards, in Chapter 7, the integration

into Open MPI is used to show how process-to-processor mapping quality affects actual program execution time regarding both uniform and non-uniform network communication cost. These benchmarks are performed on the *CHiC* parallel computer at Chemnitz University of Technology. Finally, Chapter 8 concludes.

Chapter 2

MPI Process Topologies

As already mentioned in the Introduction (see Section 1.2), one of the key issues of this work is the process mapping facility in MPI. To provide a better understanding of the rest of this work, this section gives a detailed overview of the process mapping facility, and everything related to this, as it is defined in the MPI standard [SOHL+98]. It is assumed that the reader is familiar with the basics of MPI. In particular, terms such as *Rank*, *Group*, and *Communicator* should be known. If not, [SOHL+98] is recommended for reference.

2.1 The Topology Mechanism

MPI's underlying concept for the process mapping functionality is the topology mechanism. A topology is an optional attribute that can be attached to an intra-communicator. Specifically, a topology serves two purposes:

- Provide a convenient naming mechanism for the communicator's processes
- Assist the runtime system in mapping the processes onto the hardware

The basic idea of the topology mechanism is the following: The logical communication pattern of the processes in a group is provided to the MPI runtime system. This pattern is application-dependent and is represented by a graph. It is called *virtual topology*. With the help of the virtual topology, the MPI system can compute a mapping of those processes onto the physical hardware in order to improve the communication performance.

How this mapping is computed, however, is not in the scope of MPI. Hence, it is up to the implementor of the MPI library to provide an appropriate solution, and thus many ignore this optimization facility by simply performing no reordering of processes (identical mapping).

Regardless of the possible performance benefits just desribed, a naming scheme of the processes in the virtual topology can be established. This may improve program readability and offers more notational power in message-passing programming.

2.2 Virtual Topologies

As stated above, the communication pattern of the processes in a group can be represented by a graph, the *virtual topology*: Its vertices stand for the processes, and the edges between vertices reflect pairs of communicating processes. Furthermore, the communication graph is considered symmetric, i.e., if an edge (u, v) connects vertex u to vertex v, then there is another edge (v, u) connecting vertex v to vertex u. Hence, the graph can be regarded as undirected.

Certainly, this representation of message passing in a program is a simplification. As as result, only *spatial* distribution of communication is taken into account, whereas *temporal* distribution of messages is ignored. However, in order to optimize communication time, minimizing contention for physical wires by messages occurring simultaneously is substantial on many systems. This necessitates knowledge of when messages occur and their resource requirements. On the other hand, this information might not be available at topology creation time.

Due to this, the MPI standard adopted the simpler alternative at the expense of room for optimization. Nevertheless, this approach allows a simpler interface that is well understood at the current time. Thus the programmer's task is to provide the typical communication requirements between processes, the virtual topology, to the MPI system. Here, the term typical leaves room for interpretation. The result is that the provided process topology may *over-* or *under-specify* the connectivity between processes at any time during program execution. Particularly, *over-specification* may lead to considering connections that only have little effect on overall communication time. Conversely, *under-specification*, i.e., omitting

connections that are essential to communication time leads to that those connections are neglected by a process mapping strategy. Hence, communication on the missing links might be less efficient. What is more, the topology attribute attached to the communicator will not provide a convenient way of naming those connections. Generally speaking, however, the topology mechanism, and with it the virtual topology approach, can be seen as a useful compromise between functionality and ease of usage. Evidence for that can be found in the remainder of this work.

In many parallel applications the process topologies used are regular ones such as rings, two- or higher-dimensional grids, or tori. For those structures that can be defined by (*i*) the number of dimensions and (*ii*) the number of processes in each dimension, the MPI standard uses the term *Cartesian Topologies*. Thus, there is a categorization into *cartesian* and general *graph topologies*. With this distinction, different functionality is offered. One advantage of this is that the MPI user is freed from creating a detailed graph structure consisting of vertices and edges for cartesian topologies. Instead, MPI defines convenience functions (Section 2.3.1) for those aims.

2.3 Topology Functions

Having discussed the basic ideas of MPI process topologies, this section describes the functions available in the MPI standard for using the MPI topology mechanism. Regarding notation and organization, the ANSI C version of those functions is used, and a categorization into cartesian and graph topology functions is applied. For the sake of clarification, some examples are given. More examples and a complete reference, however, can be found in [SOHL+98, Chapter 6].

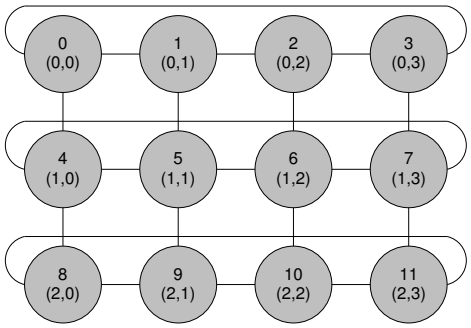


Figure 2.1: Relationship between ranks and cartesian coordinates for a 3×4 2D topology (cylinder) of processes. The upper number is the process rank, the lower value (row, column) is the coordinates.

2.3.1 Cartesian Topology Functions

According to the MPI standard, row-major numbering is used for the processes in a cartesian topology. Furthermore, the coordinates of those processes begin their numbering at 0. An example is depicted in Figure 2.1 on page 6.

Cartesian Constructor Function

```
int MPI_Cart_create (MPI_Comm comm_old,
                     int ndims,
                     int *dims,
                     int *periods,
                     int reorder,
                     MPI_Comm *comm_cart)
```

The `MPI_Cart_create()` function is collective and used to describe cartesian topologies with `ndims` dimensions. The size of each dimension is expected in the `dims` array. In `periods` it is specified, for each coordinate direction, whether the process structure is periodic or not. For instance, a 2D topology can be a rectangle, cylinder, or torus, i.e respectively, non-periodic, periodic in one dimension, or fully

periodic. The function takes an input communicator `comm_old` and returns a handle `comm_cart` to a new communicator to which the cartesian topology information is attached. Whether the MPI system is to compute an optimized mapping of the specified cartesian topology onto the underlying physical hardware, depends on the `reorder` flag. In fact, a remapping of processes is accomplished by reordering process ranks. Thus, if `reorder = 0` the rank of each process in the new group is identical to its rank in the old group. Otherwise, process ranks may be reordered in favour of an improved process-to-processor mapping.

Note that the total size of the cartesian grid can be smaller than the size of the group of `comm_old`. In that case, `MPI_COMM_NULL` is returned by `MPI_Cart_create()` for some processes, i.e. those are not in the new group. A defintion of a virtual topology with a process count larger than the size of `comm_old` results in an error.

Cartesian Convenience Function

```
int MPI_Dims_create (int nnodes,
                     int ndims,
                     int *dims)
```

`MPI_Dims_create()` is local. It helps the user compute a balanced distribution of processes per coordinate direction in a cartesian topology. Particularly, the total number of processes is expected in `nnodes`, and `ndims` provides the number of dimensions of the cartesian topology. On return, the array `dims` contains the suggested process count for each dimension, provided the dimension's entry `dims[i]` was 0. Entries not equal to 0 are assumed to be constraints on the distribution of processes, and thus are not modified. So, the user may specify the number of processes for selected dimensions in advance.

By definition, the process counts for the dimensions are as close to each other as possible. The `dims[i]` set by the call are ordered in monotonically decreasing order. A call is erroneous for negative values in `dims`, and if `nnodes` is not a multiple of $\prod_{i,\texttt{dims}[i]\neq 0} \texttt{dims}[\texttt{i}]$. Note that `dims` may be used as input to `MPI_Cart_create()`.

Cartesian Inquiry Functions

The following functions are used to inquire about the topology of a communicator. The calls are local.

```
int MPI_Cartdim_get (MPI_Comm comm,
                     int *ndims)
```

`MPI_Cartdim_get()` returns the number of dimensions of the cartesian topology associated with the communicator `comm`.

```
int MPI_Cart_get (MPI_Comm comm,
                  int maxdims,
                  int *dims,
                  int *periods,
                  int *coords)
```

The `MPI_Cart_get()` function takes a communicator `comm` and three arrays of size `maxdims`, where `maxdims` must be at least `ndims` as returned by `MPI_Cartdim_get()`. On return, `dims` and `periods` carry the number of processes and the periodicty information for each dimension of the cartesian topology associated with `comm`, respectively. `coords` yields the coordinates of the calling process in the cartesian structure.

Cartesian Translator Functions

This section discusses functions which translate process ranks into cartesian topology coordinates, and vice versa. Calls to those functions are local.

```
int MPI_Cart_rank (MPI_Comm comm,
                   int *coords,
                   int *rank)
```

The input to `MPI_Cart_rank()` is a communicator with cartesian structure `comm` and an array `coords` specifying the cartesian coordinates of a process. The result is the rank of that process in the variable `rank`. The size of `coords` must be equal to `ndims` as returned by `MPI_Cartdim_get()`.

For a periodic dimension i, the corresponding coordinate may be out of range, i.e., `coords`$[i] \notin \{0, \ldots, dims[i] - 1\}$, where $dims[i]$ denotes the size of dimension i. In that case, `coords[`i`]` is interpreted by considering periodicity and the size of dimension i. Thus, for the topology in Figure 2.1 on page 6, the coordinates $(0, 6)$ are regarded as $(0, 2)$. For non-periodic dimensions, out-of-range coordinates are erroneous.

```
int MPI_Cart_coords (MPI_Comm comm,
                     int rank,
                     int maxdims,
                     int *coords)
```

`MPI_Cart_coords()` is the rank-to-coordinates translator and can be considered the inverse operation of `MPI_Cart_rank()`. The input parameters are a cartesian topology communicator `comm`, the rank of a process within the group of `comm`, and the length of the array `coords` determined by `maxdims`. Again, `maxdims` must be at least as big as `ndims` returned by `MPI_Cartdim_get()`. On return, `coords` contains the coordinates of the process with rank `rank`.

Cartesian Shift Function

```
int MPI_Cart_shift (MPI_Comm comm,
                    int direction,
                    int disp,
                    int *rank_source,
                    int *rank_dest)
```

A cartesian shift operation is determined by the coordinate dimension of the shift and by the size of the shift step (positive or negative). The `MPI_Cart_shift()` function is local and takes a cartesian communicator `comm`, the coordinate dimension of the shift, `direction`, and the shift step size `disp`, where a positive or negative value of `disp` denotes an upwards shift or downwards shift in the corresponding coordinate, respectively. The coordinate dimensions are numbered from $0, \ldots, ndims - 1$, where $ndims$ is the number of dimensions. On return, `rank_source` and `rank_dest` contain the ranks of the two processes which are located $|$`disp`$|$ steps away from the calling process in the correponding cartesian topology dimension. Note that $|\cdot|$ yields the absolute value. Whether a process' is rank is stored in `rank_source` or `rank_dest`, depends on the shift direction (sign of `disp`).

For non-periodic dimensions, it is possible that the source and/or the destination for the shift is out of range. In that case, `rank_source` and/or `rank_dest` return `MPI_PROC_NULL`.

Two examples for Figure 2.1 on page 6 are to demonstrate the function's usage: Assume that the process with rank 0 performs two calls to `MPI_Cart_shift()`. The first call has the arguments `direction = 1` and `disp = 1`. On return, `rank_source = 3` (due to periodicity) and `rank_dest = 1`. The second call takes `direction = 0` and `disp = -2`. The result is, `rank_source = 8` and `rank_dest = MPI_PROC_NULL`. Here, the dimension 0 is non-periodic and `disp` < 0 (downwards shift).

Note that the results in `rank_source` and `rank_dest` are valid input to `MPI_Sendrecv()` (see [SOHL+98, Chapter 2]).

Cartesian Partition Function

```
int MPI_Cart_sub (MPI_Comm comm,
                  int *remain_dims,
                  MPI_comm *newcomm)
```

Having created a cartesian topology communicator with `MPI_Cart_create()`, the communicator's group can be partitioned into subgroups that form lower-dimensional cartesian subgrids, where, for each subgroup, a communicator associated with the corresponding subgrid cartesian topology is built. This functionality offers the collective call `MPI_Cart_sub()`. As its arguments, the function takes a cartesian communicator `comm` and an array `remain_dims` specifying which dimensions are to be kept in the subgrid. In particular, the i^{th} dimension is kept, if `remain_dims[`i`] = 1`. For `remain_dims[`i`] = 0`, it is dropped. On return, `newcomm` is a handle to the communicator containing the subgrid that includes the calling process.

Again, Figure 2.1 on page 6 is used for an example. Let `remain_dims = {1, 0}`. As result, 4 communicators, each with 3 processes in a one-dimensional cartesian topology, are created. Conversely, `remain_dims = {0, 1}` would create 3 communicators, each containing 4 processes arranged in a ring topology.

Cartesian Low-Level Function

```
int MPI_Cart_map (MPI_Comm comm,
                  int ndims,
                  int *dims,
                  int *periods,
                  int *newrank)
```

In case that the MPI user would like to know how the MPI system would recommend to map a certain cartesian topology onto the physical machine topology (optimized mapping), the collective `MPI_Cart_map()` function may be used. It is similar to `MPI_Cart_create()` in that it specifies an input communicator `comm` and a cartesian topology by means of the number of dimensions `ndims`, the number of processes in each dimension, `dims`, and the periodicity in each coordinate direction, `periods`. In contrast, however, reordering of process ranks is permitted implicitly. On return, instead of a new cartesian communicator, the reordered rank of the calling process is available in `newrank`. If the process does not belong to the cartesian structure `MPI_UNDEFINED` is yielded in `newrank`.

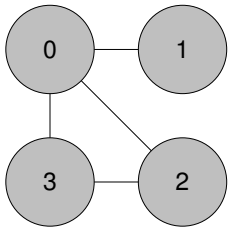
2.3.2 Graph Topology Functions

In the following, the MPI functions for creating graph topologies are discussed.

Graph Constructor Function

```
int MPI_Graph_create (MPI_Comm comm_old,
                      int nnodes,
                      int *index,
                      int *edges,
                      int reorder,
                      MPI_Comm *comm_graph)
```

The `MPI_Graph_create()` function is collective. It is used to create graph topologies. The arguments are an input communicator `comm_old`, the graph specification, and the `reorder` flag indicating whether the MPI system is to compute an optimized mapping of the graph topology onto the underlying hardware. As with cartesian topologies, a remapping of processes is accomplished by reordering process ranks. Thus, `reorder = 1` requests to reorder process ranks in favour of an optimized process-to-processor mapping. No optimization, and thus no rank reordering, is performed with `reorder = 0`. The graph structure of the virtual topology is described by means of the three parameters `nnodes`, `index` and `edges`. `nnodes` denotes the number of nodes in the graph, where they are numbered $0, \ldots,$ `nnodes` $- 1$. The array `index` describes node degrees. That means, `index[`i`]` contains the sum of the number of neighbours of the graph nodes $0, \ldots, i$. The neighbours of the graph nodes are stored in consecutive locations in the `edges` array. An example will illustrate this. Figure 2.2 depicts a graph with its adjacency list in the table next to it. The input arguments (`nnodes`, `index`, `edges`) for creating this graph can be found in Figure 2.2, as well.



Process	Neighbours
0	1, 2, 3
1	0
2	0, 3
3	0, 2

```
nnodes  =  4
index   =  {3,4,6,8}
edges   =  {1,2,3,0,0,3,0,2}
```

Figure 2.2: Graph with adjacency list and respective input arguments

As it can be seen from this example, the `edges` array is a flattened representation of the nodes' adjacency lists and has as many entries as twice the number of graph edges in the undirected topology graph. Furthermore, the array `index` is of length `nnodes`.

Thus, in C, `index[0]` is the degree of node 0, and `index[`i`] - index[`$i-1$`]` is the degree of node i, $i = 1, \ldots,$ `nnodes` $- 1$. Accordingly, the neighbours of node 0 can be found in `edges[`j`]`, $0 \leq j <$ `index[0]`, and the neighbours of node i, $i > 0$, are stored in `edges[`j`]`, `index[`$i-1$`]` $\leq j <$ `index[`i`]`.

On return, `comm_graph` is a handle to a new communicator to which the graph topology information is attached. If the number of nodes in the topology graph specification is less than the size of the group of `comm_old` some processes return `MPI_COMM_NULL`. In contrast, a topology graph larger than the group size results in an error.

Graph Inquiry Functions

Having created a graph topology, it is possible to query the topology's attributes with the functions below. These functions are local calls.

```
int MPI_Graphdims_get (MPI_Comm comm,
                       int *nnodes,
                       int *nedges)
```

`MPI_Graphdims_get()` takes a communicator `comm`, equipped with graph topology information, as input argument. On return, `nnodes` and `nedges` yield the number of nodes and the number of edges in the

graph topology, respectively. Note that the number of nodes is equal to the size of the group of `comm`. Regarding the example graph topology in Figure 2.2, the results are `nnodes = 4` and `nedges = 8`. This shows that `nedges` denotes the length of the `edges` array as supplied to `MPI_Graph_create()`.

```
int MPI_Graph_get (MPI_Comm comm,
                   int maxindex,
                   int maxedges,
                   int *index,
                   int *edges)
```

`MPI_Graph_get()` initializes the arrays `index` and `edges` as they were used by `MPI_Graph_create()` to create the communicator `comm`. The memory for these arrays has to be allocated in advance. `maxindex` and `maxedges` represent the sizes of the arrays `index` and `edges`, respectively. Here, `maxindex` has to be greater than or equal to `nnodes`, and `maxedges` must be at least as big as `nedges`, where `nnodes` and `nedges` are the results of a previous call to `MPI_Graphdims_get()`. Figure 2.2 shows a graph and the contents of `index` and `edges` after a call to `MPI_Graph_get()`, for a communicator with that graph topology.

Graph Information Functions

The following functions can be used to inquire detailed information about the structure of the graph topology. All calls are local.

```
int MPI_Graph_neighbors_count (MPI_Comm comm,
                               int rank,
                               int *nneighbors)
```

The `MPI_Graph_neighbors_count()` function returns the number of neighbours (`nneighbors`) for the process with rank `rank` in the graph topology communicator `comm`.

```
int MPI_Graph_neighbors (MPI_Comm comm,
                         int rank,
                         int maxneighbors,
                         int *neighbors)
```

`MPI_Graph_neighbors()` returns the neighbours (`neighbors`) of the process with rank `rank` in the graph topology communicator `comm`. `maxneighbors` denotes the size of the `neighbors` array. To obtain the correct size for the `neighbors` array and thus `maxneighbors`, the above `MPI_Graph_neighbors_count()` may be used. For Figure 2.2, `rank = 2` yields `neighbors = {0, 3}`.

Low-Level Graph Functions

```
int MPI_Graph_map (MPI_Comm comm,
                   int nnodes,
                   int *index,
                   int *edges,
                   int *newrank)
```

The `MPI_Graph_map()` function is the graph topology counterpart to the cartesian low-level function `MPI_Cart_map()` on page 9. For detailed information about the graph structure arguments (`nnodes`, `index`, `edges`), see `MPI_Graph_create()` on page 10. As with the cartesian low-level function, this call is collective.

2.3.3 Topology Inquiry Function

Sometimes it may be necessary to query what type of topology a communicator is associated with. In this case, `MPI_Topo_test()` yields the information required.

```
int MPI_Topo_test (MPI_Comm comm,
                   int *status
```

`MPI_Topo_test()` takes a communicator `comm` as input argument and returns the communicator's topology type in `status`. The possible values of `status` and their respective meanings are depicted in Table 2.1 below. This call is local.

status	Description
MPI_GRAPH	Graph topology
MPI_CART	Cartesian topology
MPI_UNDEFINED	No topology

Table 2.1: Topology types

Chapter 3

The Mapping Problem

Having discussed the MPI process topology mechanism as it is defined in the MPI standard [SOHL+98] in the previous chapter, this chapter investigates the process-to-processor mapping problem with respect to the MPI process mapping facility. In particular, different approaches for evaluating the quality of mappings are considered and, based on this, a formal description of the mapping problem is given.

3.1 Introduction

As already mentioned (Section 2.2 on page 5), in MPI, the communication pattern of the processes in a group can be described by means of a graph $G_v = (V_v, E_v)$, the virtual topology. The processes are represented by the set of vertices V_v and the edges in E_v denote the pairs of primarily communicating processes. Similarly, the underlying network topology of a parallel computer can be described by a graph $G_n = (V_n, E_n)$, where V_n is the processors and E_n is the network links between processors. As stated in Section 1.2, in this work, the number of processes $|V_v|$ is assumed to be greater than the number of processors $|V_n|$. Additionally, two functions are given: $w_n : V_n \rightarrow \mathbb{N}\backslash\{0\}$ and $c_n : V_n \times V_n \rightarrow \mathbb{R}$. The former, $w_n(v)$, denotes how many processes have to be assigned to processor v, and the latter, $c_n(u, v)$, states the cost to communicate between the processors u and v.

Note that c_n is not the adjacency matrix representation of the network topology graph G_n. It is rather a function that yields the cost to communicate for all pairs of processors. That means, the construction of c_n may be influenced by the network topology graph G_n, but does not necessarily have to be. Criteria for an appropriate choice of c_n are discussed below (Section 3.2).

Finally, the objective is to find a mapping $\pi : V_v \rightarrow V_n$ which satisfies the size constraints w_n and minimizes a mapping cost function. The purpose of such a mapping cost function is to have a means by which the quality of different mappings can be compared for the same input (virtual topology graph G_v, network topology graph G_n, w_n, and c_n).

3.2 Modelling Communication Cost between Processors

From the definition above, c_n can be regarded as a square matrix of size $|V_n| \times |V_n|$. Thus, every pair of processors is assigned a real number. Note that this matrix does not have to be symmetric. This means that the cost for sending data from processor u to processor v may differ from sending data in the opposite direction, i.e. from v to u. What is more, the function c_n is also defined for $c_n(u, u)$, $u \in V_n$, i.e. the sender is equal to the receiver. In this case and under the assumption that, compared to inter-processor communication cost, intra-processor communication cost is negligible, it is reasonable to yield the constant 0. Thus, the main diagonal entries in the matrix representation of c_n are set to 0. However, regarding the choice of the remaining matrix coefficients, some questions arise. For instance,

- How to model network communication cost for a pair of processors by *one* parameter only?
- How to take network latency and bandwidth into account and how to weigh them?
- What about network contention?

In the following, these questions are addressed and advantages and disadvantages of the matrix approach are examined.

Basically, the term communication cost is rather abstract. The idea of communication cost in connection with the process-to-processor mapping problem is to have a measure by which the time required for communicating a message between any pair of processors can be estimated. Based on this measure, all $|V_n|^2$ communication channels between all pairs of processors (considering order and including the cases where sender is equal to receiver) can be compared according to their performance. This enables a mapping strategy to compute mappings that favour fast communication channels. As a consequence,

communicating processes (according to the virtual topology graph G_v) are either mapped onto the same processor, or onto processors with fast communication channels between each other.

Certainly, a fixed scalar per communication channel does not seem to be sufficient to model an interconnection network. The main reason is that network performance is not only determined by parameters such as latency and bandwidth, but also by network contention, a highly dynamic parameter which is also affected by communication caused by entirely independet applications competing for the network resources. Hence, the communication cost matrix approach is, at best, a rough approximation. On the other hand, it must be noted that an appropriate mapping cost function should also, based on the current mapping, consider the communication cost matrix for evaluating a mapping's quality. Hence, a too complex method for modelling communication cost between processors might increase the running time of the mapping optimization process too much. All in all, however, the cost matrix is intuitive and, with the "right" values, a powerful tool. The sections below discuss ideas for obtaining the remaining coefficients in the communication cost matrix representation of c_n.

3.2.1 Hop Metric Model

One possibility for constructing c_n is to use the hop metric. This means that $c_n(u, v)$ yields the number of network links a message has to travel on a path between processor u and processor v. Note that this path and thus the hop count is determined by the network's routing algorithm. The hop metric is similar to the path length network model in [WC98]. Generally speaking, modelling communication cost by means of the hop metric is more appropriate for interconnection networks with static routes between processors. Otherwise, adaptive routing complicates the process of assigning a hop count to a communication channel between two processors.

As stated above, the hop metric is based on the network topology and the routing algorithm exclusively. For that reason, it is oblivious to physical parameters such as latency and bandwidth of the interconnect. However, the hop count between processors can be regarded as a measure which is closely related to latency, since a long path corresponds to an increase in latency in comparison to a short path. Furthermore, with reference to networks with different network technologies and thus differing performance parameters, as can be found in grid computing, the simple hop metric approach as described above would not be appropriate for modelling communication cost. This is due to that, for instance, a wide area network (WAN) link is much slower than a high speed system area (SAN) network link and thus they should not be considered equal.

3.2.2 Parallel Computational Models

Another method for modelling communication cost between processors is to abstract the internal structure of the communication network (e.g. network topology) by means of a few performance parameters. This is the way that some computational models go.

Parallel computational models try to capture the execution characteristics (performance) of actual parallel machines with the help of a set of parameters. This is done in order to have a machine model based on which the running time of parallel algorithms can be predicted. Thus, a programmer is able to analyse their algorithm and to reveal performance bottlenecks. Additionally, different algorithms can be compared on the basis of a uniform machine model more easily. The difficulty in developing parallel computational models is to find a trade-off between the degree of detail, i.e. its accuracy, and the tractability of the model. In the following, some models for parallel computation are discussed in respect of their applicability to modelling communication cost between processors.

LogP Model

The LogP model [CKP+93] describes a parallel machine and the performance of its interconnection network by means of the parameters below. Note that processors are assumed to work asynchronously and communicate by fixed short size point-to-point messages.

- **Latency, L:** An upper bound on the latency, incurred by sending a short message from its source processor to its target processor.
- **Overhead, o:** The length of time that a processor is engaged in the transmission or reception of each message. During this time, the processor cannot perform other operations.

- **Gap between messages, g:** The minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal $1/g$ corresponds to the available per processor communication bandwidth for short messages.
- **Number of processors, P:** The number of processors.

The parameters L, o, and g are assumed to be measured as multiples of processor cycles. Based on this model, the time required for sending a small message between two processors is: $o + L + o$. Particularly, o cycles on the sender, L cycles for communication latency, and additional o cycles on the receiving processor. Similarly, sending k small consecutive messages takes $o + (k-1) \cdot \max\{g, o\} + L + o$ cycles. What is more, a network's finite capacity is modelled. This means that at most $\lceil L/g \rceil$ messages can be in transit to any processor or from any processor at any point in time.

As it can be seen from the model's parameters, network characteristics such as latency (L) and bandwidth ($1/g$) are taken into account. Hence, the LogP model better reflects actual network performance compared to the hop metric which only considers topology information. However, identical values for the 2 network parameters (L, g) are used to compute communication time between any pair of processor. Hence, variation in L or g over the parallel machine leads to imprecise running time predictions of a parallel program and, what is more important, does not model variation in communication cost between processors in a parallel system, adequately.

LogGP Model

An extension of the LogP model was proposed by Alexandrov et al in [AISS97]. It is called LogGP and adds support for modelling long messages. The new parameter is:

- **Gap per byte, G:** The gap per byte for long messages is defined as the time per byte for a long message. The reciprocal $1/G$ corresponds to the available per processor communication bandwidth for long messages.

The remaining parameters L, o, g, and P are taken from the LogP model. Under LogGP, the latency L is the same for short and long messages.

Regarding communication times, the time for transmitting a small message can be analyzed as in the LogP model. Under LogGP, sending k bytes as a single long message takes $o + (k-1) \cdot G + L + o$ cycles: First, o cycles of overhead are required at the sender to get the first byte into the network. Each subsequent byte takes G cycles to go out. At time $o + (k-1) \cdot G$, the last byte goes into the network. To arrive at the destination processor, each byte has to travel through the network for L cycles. In consequence, the last byte exits the network at time $o + (k-1) \cdot G + L$. Finally, the target processor is engaged in the reception of the message for o cycles. All in all, the entire message is available at the receiver after $o + (k-1) \cdot G + L + o$ cycles.

In contrast, however, sending a k byte message under the LogP model requires sending $\lceil k/w \rceil$ messages. Hence, $o + (\lceil k/w \rceil - 1) \cdot \max\{g, o\} + L + o$ cycles are necessary. Note that w is the underlying message size of the parallel machine.

With reference to modelling communication cost, the advantage of the LogGP model over the LogP model is that it more accurately models the time required for sending long messages. This is necessary since state-of-the-art interconnects provide special support (e.g. bulk transfers) for long messages with an increase in bandwidth compared to short messages. However, the question which arises with this more accurate approach is, what message size to use to model communication cost. In other words, how to weigh the latency (L) and bandwidth ($1/G$) parameters of a network. An appropriate message size should correspond to the parallel application's (most frequent) message size, as close as possible. Unfortunately, this is not always known and may require an in-depth analysis of the program's communication pattern. Furthermore, as with the LogP model, the LogGP model does not consider variations in the network performance parameters (L, g, G) and thus is not aware of different communication cost between different processors.

HLogGP Model

In order to address heterogeneity both in the computational nodes and in the communication network, the heterogeneous parallel computational model HLogGP was proposed by J. L. Bosque and L. P. Perez in [BP04]. It is based on the LogGP model which is extended to deal with heterogeneous systems. In this respect, the term *heterogeneous* is equivalent in meaning to *difference in performance*. The idea is

to replace LogGP's scalar parameters (L, o, g, G, P) by vector or matrix parameters. The description of the proposed parameters is based on an M nodes cluster $(N_1, \ldots, N_M)$. Particularly, the modified parameters are:

- **Latency, L:** Communication latency. The latency is affected by network technology and topology (routing delays). In comparison to LogGP, L is a square matrix denoting the *Latency Matrix* of a heterogeneous cluster. Hence, $L = \{l_{1,1}, \ldots, l_{M,M}\}$, where $l_{i,j}$ represents the latency in sending a message from node N_i and to node N_j.

- **Overhead, o:** The time that a processor needs to send or receive a message. The operations for sending or receiving a message are different. For that reason, different amounts of overhead occur. To take this into account, two independent overhead parameters O_s (sender overhead) and O_r (receiver overhead) are proposed. Both O_s and O_r depend on each node's computational power. Thus, instead of scalar parameters, vector parameters are needed in order to account for variation in computational power among different nodes. Finally, the *sender overhead vector* is defined as $O_s = \{o_{s_1}, \ldots, o_{s_M}\}$ and the *receiver overhead vector* as $O_r = \{o_{r_1}, \ldots, o_{r_M}\}$, where o_{s_i} and o_{r_i} denote the sender and receiver overhead for node N_i, respectively.

- **Gap between messages, g:** This parameter depends on how fast a node can send consecutive short messages. It is determined by the node's network interface and thus is a node feature. In consequence, the gap is defined as a vector $g = \{g_1, \ldots, g_M\}$, the *gap vector*. Note that g_i represents node N_i's gap for one byte long messages.

- **Gap per byte, G:** The gap per byte depends on the network technology's capabilities to manage long messages and is determined by the network bandwidth. Network bandwidth, however, is influenced both by network technology and topology. Thus, the gap per byte for a particular message depends on its path to the destination processor. As a result, a *Gap Matrix* $G = \{G_{1,1}, \ldots, G_{M,M}\}$ is defined in which an element $G_{i,j}$ records the gap per byte for communicating a long message from node N_i to node N_j. The inverse $1/G_{i,j}$ is the bandwidth available for sending long messages between the corresponding pair of nodes.

- **Computational power, $\mathbf{P_i}$:** Since each node may have very different computational power, the number of nodes P in the LogGP model is not suitable for modelling a system's computational power. Instead, each node N_i is assigned a value P_i denoting its computational power. Here, computational power can be defined as the amount of work finished by a processor during a unit time span for a specific application. Regarding this definition, P_i depends on the node's physical features (CPU organization and speed, memory and I/O capabilities, etc.) as well as on the implementation of the algorithm which is executed. Finally, a *computational power vector* $P = \{P_1, \ldots, P_M\}$ is introduced which stores the computational power P_i of node N_i.

The advantage of the HLogGP model is that it accounts for differences in the communication performance between different pairs of processors. Due to this, it is much more appropriate for modelling communication cost between processors. However, regarding the assessment of the HLogGP parameters, a concern is the amount of parameters which are required with an increasing number of processors. For instance, in a simple approach, all pairs of processors need to communicate with each other in order to calculate the *Latency Matrix* L and the *Gap Matrix* G. This produces much contention in the communication network and may drastically denigrate communication performance of other independent parallel jobs on the same machine. Furthermore, due to much network load, parameter values could be calculated which are not realistic (too pessimistic) under network load conditions found during the actual program execution. One way to circumvent this is to perform the parameter assessment for each pair of processors one after the other, sequentially. However, this can be very time consuming and, in this particular case, parameter assessment could be too optimistic (due to little, artificially enforced, network traffic). For these reasons, network contention is another factor that complicates parameter assessment for computational models, such as HLogGP, and thus the calculation of the coefficients for the communication cost matrix.

LoGPC Model

In order to account for the impact of network contention on parallel programs' execution times, C. A. Moritz and M. I. Frank proposed the LoGPC [MF01] machine model. It is based on the LogP and LogGP

models and also considers the influence of a network interface's DMA (Direct Memory Access) unit on communication performance.

Under the assumption of contention-free communication, for short messages, the LoGPC model uses the same parameter set as the LogP model. Similarly, for long messages, the parameters of the LogGP model are used and, in addition, an extension considering DMA memory transfers by the network interface.

In [MF01], the underlying network is assumed to be a k-ary n-cube with wormhole routing. Additionally, in order to model contention delays observed by a particular program in the communication network, both (i) the LogGP machine parameters, and (ii) information about the program's messaging rate is needed. Finally, (i) and (ii) are applied to a queueing model. In consequence, considering contention delay, the end-to-end message delivery time required for sending a short message is

$$T_{s-r} = o_s + L + C_n + o_r \,,$$

where

$$C_n = \frac{(n+1)(k_d - 1) \cdot B^2 \cdot m_c/2}{1 - m_c \cdot B \cdot k_d/2} \tag{3.1}$$

with

$$m_c = \frac{1}{T + C_n}. \tag{3.2}$$

Table 3.1 describes the meanings of the variables in the equations above.

With respect to long messages of size B bytes, the time at which the complete message is available at the receiver is

$$T_{s-r} = o_{sl} + (B-1) \cdot G + L + C_n \,, \tag{3.3}$$

with C_n and m_c as in the Equations (3.1) and (3.2), respectively. A description of the variables can be found in Table 3.1, as well. Note that, in comparison to the LogGP model, the end-to-end delivery time for long messages in Equation (3.3) considers pipelining in the network interface's DMA unit. More details can be found in [MF01].

In order to calculate the actual contention delay per message C_n, firstly, Equation (3.2) has to be solved after substituting (3.1) for C_n. In other words, calculate m_c by solving the quadratic equation (quadratic in m_c)

$$m_c = \frac{1}{T + \frac{(n+1)(k_d-1)\cdot B^2 \cdot m_c/2}{1 - m_c \cdot B \cdot k_d/2}} \tag{3.4}$$

by means of the machine parameters and the information about the parallel program's communication pattern. Finally, replace m_c by its result in (3.1). The calculations above are presented in order to show the information on which the contention delay computation is based. More details, however, can be found in [MF01].

As it can be seen from the discussion above, under the LoGPC model, calculating contention delay for a parallel application requires additional information or assumptions:

1. The network topology is assumed to be a k-ary n-cube with wormhole routing.
2. The information about the average distance, k_d, a message travels in each dimension in the network. This parameter is implicitly based on a process-to-processor mapping for the given application.
3. The knowledge about the application's most frequently sent message size B.
4. The information about the application's average time between messages T, regardless of network contention.

To obtain this information, a deeper knowledge about both the parallel system and the parallel program's communication pattern is necessary. For that reason, the LogPC model seems to be too complex for creating a communication cost matrix for modelling communication performance between the processors in a network, based on the contention delays produced by a parallel program. Another issue is that the contention calculation in the LoGPC model is based on the communication characteristics of one application only. Thus, network traffic caused by other jobs on a parallel machine is not taken into consideration. Certainly, this would make the parallel machine model even more complex. Nevertheless, it should be mentioned in the context of modelling communication cost between processors for optimizing process-to-processor mappings.

Variable	Description
o_s	Send overhead for short messages
o_{sl}	Send overhead for long messages
o_r	Receive overhead for short messages
T_{s-r}	End-to-end message delivery time
L	Upper bound on network latency without contention
C_n	Network contention delay per message travelling an average distance of k_d in each of the n network dimensions
B	Message size in bytes
G	Gap per byte on the network for long messages
T	Application's average time between messages without contention
$1/T$	Application's message rate without contention
n	Dimension of k-ary n-cube network
k_d	Average distance a message travels in each network dimension
m_c	Application's average message injection rate into the network with contention

Table 3.1: Description of LoGPC parameters

3.2.3 Conclusion

Having discussed some aspects of different approaches for computing the remaining coefficients in the communication cost matrix, it mainly depends on factors such as the information available about the parallel system and the application, the accuracy required, and, equally important, the time available for parameter assessment, which methodology to choose. Nevertheless, the intention of the communication cost function c_n, in conjunction with a mapping cost function, is to guide a process-to-processor mapping strategy in computing a mapping which prefers communication locality, i.e., communicating processes are mapped onto nearby processors. The reason for this is that reducing the average distance that messages have to travel improves latency and saves bandwidth since it reduces both the number of hops per message and the number of messages in the network competing for resources.

Here, the emphasis is on *guiding* a mapping algorithm. This is due to that C. Walshaw and M. Cross [WC98] showed that applying a quadratic path length network model (i.e. communication cost between any two processors is the lengh of the path between those processors to the power of 2) for modelling network communication cost yields mappings with more communication locality than the linear path length model. However, penalizing non-local communication to much, for instance, with a cubic path length network model, can lead to mappings that move communication into the interconnection network (in the case where each processor is assigned more than one process), in order to avoid additional cost caused by longer message paths.

3.3 Mapping Cost Functions

This section discusses mapping cost functions and their suitability for assessing the quality of process-to-processor mappings with respect to communication overhead caused by inter-processor communication. In order to optimize communication time of a parallel program, the mapping cost function must be chosen carefully. In general, communication time in a parallel system with processors of equal computational power is mainly determined by:

- Latency, bandwidth, and contention in the interconnection network.
- Size of the messages and the rate at which an application injects messages into the network.
- Number of hops that each message has to travel.

Given a specific application which is to be executed on a specific parallel computer, the only two parameters, from those above, that can be influenced is network contention and the number of hops per message. Hence, the following metrics for mapping cost functions seem to be appropriate for modelling communication cost occurring during a program run, on the basis of a certain process-to-processor mapping. For the sake of completeness, a new measure $c_v(s,t)$ is introduced. It accounts for the amount of communication required between two processes s and t.

3.3.1 Edge Cut (Φ)

In its most general form, the edge cut is the total weight of cut edges, i.e., communication requirements between processes that are mapped onto different processors.

$$\Phi = \sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} c_v(u,v)$$

Thus, it accounts for the success of hiding communication in processors (intra-processor communication). On the other hand, the edge cut does not consider differences in communication cost between processors (Section 3.2). For that reason, it is only suitable for modelling uniform communication cost.

In connection with the MPI mapping problem, the edge cut measure is reduced to

$$\Phi' = \sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} 1\,.$$

This is due to the fact that the virtual topology graph (Section 2.2), as defined in MPI, is unweighted. Thus, in this context, the edge cut counts the number of edges from the virtual topology graph G_v whose vertices are assigned to different processors under a mapping π.

3.3.2 Weighted Edge Cut (Γ)

An extension of the edge cut Φ leads to the weighted edge cut. That means, every communication requirement between two processes is weighted by a factor denoting the cost to communicate between the two processors (Section 3.2) which the two processes are mapped onto. In other words, the weighted edge cut measure favours intra-processor communication and considers non-uniform network communication cost. Thus,

$$\Gamma = \sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} c_v(u,v) \cdot c_n(\pi(u), \pi(v))\,,$$

and in terms of MPI,

$$\Gamma' = \sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} c_n(\pi(u), \pi(v))\,,$$

since G_v is unweighted and c_n is not in the scope of the MPI standard [SOHL+98] (i.e., an MPI implementation may consider non-uniform communication cost in an interconnection network during process-to-processor mapping optimization, see Section 3.2).

3.3.3 Average Dilation (Δ)

In order to define the average dilation, a function $hops_n(u,v)$ is introduced. It yields the number of hops in the network between two processors u and v. With this new function, the average dilation measure can be defined as

$$\Delta = \frac{\sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} hops_n(\pi(u), \pi(v))}{\sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} 1},$$

i.e., the average number of hops a message has to travel according to a mapping π. Thereby, only messages travelling between, not within, processors are considered. Furthermore, under the assumption that c_n is equal to $hops_n$, the average dilation can also be written as

$$\Delta = \frac{\sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} c_n(\pi(u), \pi(v))}{\sum_{\substack{\{u,v\} \in E_v \\ \pi(u) \neq \pi(v)}} 1} = \frac{\Gamma'}{\Phi'}.$$

All in all, the average dilation measure favours mapping communicating processes onto nearby processors while ignoring intra-processor communication. As a result, the situation as in Figure 3.1 might occur. In particular, two different mappings of a 1D grid of four processes onto two processors P_0, P_1 are given. Both mappings are considered the same quality according to the definition of the average dilation Δ. However, practically, they are not equivalent. The reason is that the first mapping causes more inter-processor communication than the second one.

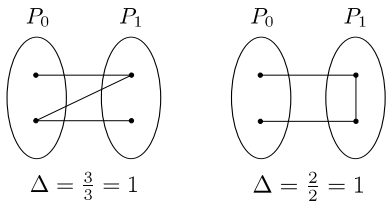


Figure 3.1: Average dilation shortcoming

3.3.4 Conclusion

Based on the discussion above and the fact that $|V_v| > |V_n|$ is assumed in this work, the weighted edge cut Γ is the cost function of choice. In [BGH+05] and [MYY+01] the weighted edge cut was used, as well. In addition, [BGH+05] showed its practical relevance to modelling communication time.

Nevertheless, another criterion for a mapping cost function could be to also consider the maximum amount of inter-processor communication that a single processor has to manage [HK00]. This means that, according to a process-to-processor mapping, the number of edges (communication requirements between processes) that logically connect processors with each other should be equally distributed among all the processors executing the parallel program. The reason for this is that total program execution time is determined by the slowest processor. Hence, similarly to balancing computation among processors, communication should be balanced, as well. Otherwise, a single processor may become the communication bottleneck during program execution.

3.4 Mapping Problem Formalization

By means of the information given so far, it is possible to give a more formal description of the mapping problem as it applies to the MPI process mapping facility (Section 2). In the following, two variants of the mapping problem are given, one for networks with uniform communication cost and the other for interconnection networks with non-uniform communication cost.

Considering networks with uniform communication cost leads to a variant of the k-way graph partitioning problem: Given a graph $G_v = (V_v, E_v)$ with $|V_v|$ vertices, the number of subsets $k \in \mathbb{N}\backslash\{0\}$, and k subset sizes $s_1, \ldots, s_k \in \mathbb{N}\backslash\{0\}$ with $\sum_{i=1}^{k} s_i = |V_v|$. Find a partition π, $\pi : V_v \rightarrow \{V_1, \ldots, V_k\}$, of V_v into k disjoint sets $V_1, \ldots, V_k$ such that

$$\begin{array}{ll} (i) & |V_i| = s_i, \qquad i = 1, \ldots, k \\ (ii) & \Phi^{'} \longrightarrow \min . \end{array}$$

Here, (i) reflects the size constraints of the processors and (ii) denotes the minimization of the edge cut $\Phi^{'}$. The graph partitioning problem is known to be NP-complete [GJ79], thus it cannot exactly be solved in polynomial time. This necessitates polynomial time approximation algorithms.

Regarding networks with non-uniform communication cost between processors (Section 3.2), the mapping problem can be summarized as: Given a graph $G_n = (V_n, E_n)$ with $|V_n|$ vertices, a function $w_n : V_n \rightarrow \mathbb{N}\backslash\{0\}$, and a second function $c_n : V_n \times V_n \rightarrow \mathbb{R}$. Given a second graph $G_v = (V_v, E_v)$ with $|V_v|$ vertices. The objective is to find a function π, $\pi : V_v \rightarrow V_n$, that assigns every vertex in V_v to any vertex in V_n such that

$$\begin{array}{ll} (i) & |\{j \in V_v \,|\, \pi(j) = i\}| = w_n(i), \quad i = 1, \ldots, |V_n| \\ (ii) & \Gamma^{'} \longrightarrow \min . \end{array}$$

Similarly as above, (i) guarantees to satisfy the size constraints of the processors, and (ii) stands for the minimization of the weighted edge cut $\Gamma^{'}$. It is obvious that the mapping problem is a generalization of the graph partitioning problem and it can be shown that it is NP-complete, as well. In consequence, mapping heuristics which approximate good solutions in polynomial time are needed.

Chapter 4

Mapping Strategies

This chapter investigates different mapping strategies in connection with the MPI mapping problem as discussed in the previous chapter. The first section gives a short introduction. Then, after an overview about basic techniques which are used in state-of-the-art mapping strategies, the strategies considered in this work are discussed in more detail.

4.1 Introduction

Due to the mapping problem's computational complexity, only optimal solutions for special cases can be found efficiently. Hence, in the following, the focus is on mapping heuristics. Furthermore, as already mentioned in The Scope of this Work (Section 1.2), these heuristics are generic in the sense that they can be used to compute process-to-processor mappings for any desired combination of virtual topology and network topology. Thereby, however, most strategies yield better results for certain combinations than for others.

For the sake of convenience, all the methods below are assumed to get the same input, i.e., a virtual topology graph $G_v = (V_v, E_v)$ with the processes numbered $0, \ldots, |V_v| - 1$, and a network topology graph $G_n = (V_n, E_n)$ with the processors $0, \ldots, |V_n| - 1$. Note that the functions w_n and c_n (Section 3.1) are not necessary for the algorithm descriptions below.

4.2 Basic Techniques and Approaches

For a better understanding of the mapping algorithms in Section 4.3, this section describes basic techniques that are applied in those methods.

4.2.1 Multilevel Graph Partitioning

Basically, the idea of the multilevel graph partitioning approach is to create a sequence of increasingly smaller graphs that approximate the original graph, partition the smallest graph, and project the partitioning back through the intermediate levels. This approach yields the following three phases that are depicted in Figure 4.1.

- **Coarsening phase:** A sequence of smaller graphs $G_1, G_2, \ldots, G_m$ is constructed from the original graph G_0, such that $|V_0| > |V_1| > |V_2| > \cdots > |V_m|$.
- **Partitioning phase:** An initial partition P_m of G_m is computed.
- **Refinement phase:** The final partitioning P_0 of G_0 is obtained by projecting P_m back through the intermediate partitionings $P_{m-1}, P_{m-2}, \ldots, P_1$. During the projectioning the resulting partitionings are refined at each level.

4.2.2 Graph Contraction

The graph contraction operation is a fundamental part in the multilevel graph partitioning approach (Section 4.2.1). It constructs a smaller graph from the original graph by means of collapsing edges in the original graph. This can be achieved in two steps:

1. A *matching* is computed, i.e., a set of edges where no two edges are incident on the same vertex. Since the purpose of graph contraction is to reduce the size of a graph, mostly a *maximal matching* is preferred. A matching is maximal if no additional edges can be added without offending the matching property.

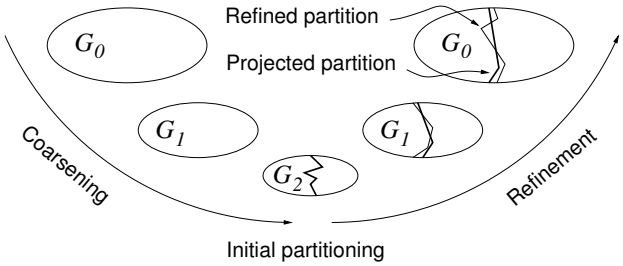


Figure 4.1: The multilevel (2-way) graph partitioning approach

2. Each edge in the matching is contracted, i.e., the two vertices i, j that are part of the edge are combined to form a new vertex v. The weight of v is the sum of the weights of the vertices i and j. The edges of v are the union of the edges of i and j. In case that i and j are adjacent to the same vertex k, then the weight of the edge $\{v, k\}$ is the sum of the weights of the edges $\{i, k\}$ and $\{j, k\}$.

4.2.3 Kernighan-Lin Heuristics

A class of algorithms that are often used to refine graph partitionings in the multilevel refinement phase (Section 4.2.1) is the Kernighan-Lin (KL) heuristics. Originally, the idea by Kernighan and Lin [KL72] for graph bipartitioning is the following: A greedy method starts with an arbitrary bipartitioning of the graph. In each iteration is searched for two subsets of vertices, one from each part of the graph, such that swapping them yields a better partition. If those sets are found the swap is performed and the resulting bipartition becomes the input for the next iteration. In the case that no swap can be performed, a local minimum is achieved and the algorithm terminates (since no further improvement is possible with the KL algorithm).

An improvement of the KL algorithm that has almost linear time complexity is that by Fiduccia and Mattheyses (FM) [FM82]. Instead of swapping sets of vertices, it considers sequences of vertex moves between the two sets of the graph bipartitioning. By also accepting moves that reduce partition quality, the FM algorithm, in contrast to KL, is able to *climb out of local minima*.

4.3 The Mapping Strategies

This section describes the mapping strategies that are compared to each other in subsequent sections. For this reason, the implementations of some algorithms are taken from so-called graph partitioning packages such as Chaco [HL95], Metis [KK98a], Jostle [Wal02], and Scotch [Pel07]. Hence, the names of the respective packages appear in the strategy names, if necessary. Table 4.1 lists those graph partitioners with their respective version information as they are used in this work.

Graph partitioning package	Version
Chaco	2.2
Metis	4.0.1
Jostle	3.0
Scotch	5.0.3

Table 4.1: Gaph partitioning packages' version information

4.3.1 Chaco Linear

The linear mapping scheme, implemented in Chaco [HL95], is used as reference, since it is the most popular mapping strategy among existing MPI implementations. Let $|V_v|/|V_n|$ be an integer k and the set of processes to be equally distributed across the processors, then $\pi(i) = i \; DIV \; k$, $i \in V_v$. Here, DIV denotes the integer division. In other words, the first k processes are assigned to processor 0, the next k to processor 1, etc. This yields a mapping that is neither aware of topology information nor of any machine parameters (Section 3.2). However, it might be surprisingly good because neighbouring processes in a

regular topology (as focused on in this work) are usually assigned numbers of similar size and thus are mapped onto the same or a nearby processor.

Regarding the complexity of the linear scheme, the corresponding processor for a single process is computed by means of a constant number of operations (one integer division). Since every process is considered once, the computational complexity is in $O(|V_v|)$.

4.3.2 Chaco MLRB

The Chaco multilevel recursive bisectioning (MLRB) algorithm [HL93] is a k-way graph partitioning method that partitions the graph G_v into $|V_n|$ parts by means of recursive bisectioning of G_v. In every bisectioning step a multilevel 2-way graph partitioning algorithm is used (Section 4.2.1). Regarding the multilevel partitioning approach, Chaco MLRB uses random matching (RM) in the coarsening phase, spectral bisectioning for the initial partitioning, and the repetitive refinement of the partitioning is accomplished by a variant of the Fiduccia and Mattheyses (FM) strategy. The complexity of the Chaco MLRB strategy is mainly determined by the multilevel 2-way graph partitioning method and the depth of the recursive bisectioning tree. Since the recursion tree is of depth $O(\log |V_n|)$ and each multilevel graph bisectioning costs $O(|E_v|)$, the overall asymptotic running time is of order $O(|E_v| \log |V_n|)$.

4.3.3 Metis MLRB

This strategy [KK99] is similar to Chaco MLRB in that it is a multilevel k-way graph partitioning algorithm which is based on recursive bisectioning. However, the main differences can be found in the multilevel phases: The coarsening is performed by a variant of heavy edge matching (HEM), the initial partitioning by region growing, and, in the refinement phase, a variant of FM is applied. As with Chaco MLRB, the complexity is $O(|E_v| \log |V_n|)$.

4.3.4 Metis MLkP

Metis MLkP [KK98b] is a multilevel k-way partitioning algorithm which, instead of recursive bisectioning, directly creates a k-way partitioning. This affects the multilevel partitioning scheme in that the initial partition is a k-way partition and, in the refinement phase, this k-way partition, instead of a bisection, has to be refined. In particular, the algorithms used for coarsening, initial partitioning, and refinement are, respectively, a variant of HEM, the Metis MLRB method described above, and random boundary refinement. Regarding the running time, the contraction phase coarsens the graph G_v until $O(|V_n|)$ vertices are left. This requires $O(|E_v|)$ operations. Then, under the assumption that the coarsest graph has $O(|V_n|)$ edges, an initial $|V_n|$-way partitioning can be computed in time $O(|V_n| \log |V_n|)$ by means of the Metis MLRB method. Furthermore, in [KK98b], it is assumed that $O(|V_n| \log |V_n|)$ is often smaller than $O(|E_v|)$. Hence, the initial partitioning requires $O(|E_v|)$ operations. Afterwards, the refinement phase has complexity $O(|E_v|)$. As consequence, the overall complexity of Metis MLkP is $O(|E_v|)$.

4.3.5 Jostle MLkP

Similarly to Metis MLkP, the Jostle MLkP [WC98] mapping strategy is a multilevel k-way partitioning scheme based on direct k-way partitioning. In particular, the original graph G_v is successively coarsened with HEM until the number of vertices is equal to the number of processors $|V_n|$. Thereafter, the initial partitioning assigns vertex i from the coarsest graph to vertex i in the network topology graph G_n. During the refinement phase the intermediate k-way partitionings are improved by means of an FM like algorithm which incorporates load balancing using the diffuse algorithm of Hu et al. [HB99]. Regarding the complexity of the Jostle MLkP mapping strategy, the authors argue that the running time varies very much, dependent on the input. Since no complexity analysis can be found in [WC98] and a worst case analysis might be too pessimistic in practice, it is omitted. Nevertheless, to get an idea of the method's efficiency, the benchmarks below show absolute running times for some examples.

4.3.6 Jostle MLkP+

This strategy [WC01] is an extension of the Jostle MLkP method in order to account for non-uniform communication cost. This means that the coarsening phase (with HEM) is left unchanged. However, during initial partitioning the $|V_n|$ vertices in the coarsest graph are mapped onto the $|V_n|$ vertices in the network topology graph so as to minimize communication cost (in a network with non-uniform

communication cost). C. Walshaw and M. Cross formulate this as an instance of the quadratic assignment problem (QAP) [FBR87] which is NP-complete. For that reason, they use a heuristic algorithm [BR84] based on simulated annealing (SA) to approximate a good initial partitioning. Finally, the refinement algorithm from JOSTLE MLkP (FM like) is adapted to consider non-uniform network communication cost. In respect of the running time, the same arguments as with JOSTLE MLkP apply to JOSTLE MLkP+.

4.3.7 SCOTCH DRB

The dual recursive bipartitioning (DRB) algorithm [PR96], implemented in SCOTCH, is a mapping strategy taking non-uniform communication cost into account. It is based on the divide-and-conquer approach. Initially, the set of processors V_n, called domain, is associated with the set of processes V_v. At each step, the algorithm bipartitions a domain of processors into two disjoint subdomains and calls a graph bipartitioning algorithm to distribute the subset of processes associated with that domain across the two subdomains. The recursion stops if a process subset is empty or if a processor domain consists of a single processor. In the latter case, the processes associated with this domain are assigned to the single processor. See Listing 4.1 for a pseudocode description.

```
mapping (D, P)
Set_of_processors  D;
Set_of_processes   P;
{
  Set_of_processors  D0, D1;
  Set_of_processes   P0, P1;

  if (|P| == 0)    /* If nothing to do. */
    return;

  if (|D| == 1) { /* If one processor in D */
    result (D, P); /* P is mapped onto it. */
    return;
  }

  (D0, D1) = processor_bipartition (D);
  (P0, P1) = process_bipartition    (P, D0, D1);

  mapping (D0, P0); /* Perform recursion. */
  mapping (D1, P1);
}
```

Listing 4.1: SCOTCH DRB algorithm pseudocode ([Pel07])

Thus, the algorithm computes partial mappings which become more and more detailed during the recursive descent. A complete mapping is achieved when repeated bipartitioning has reduced all processor subdomain sizes to one.

For processor subdomain bipartitioning, SCOTCH offers two different approaches. In the first one, processor subdomains are bipartitioned by means of a usual graph bipartitioning algorithm (domain decomposition by partitioning). Alternatively, depending on the architecuture graph (network topology), a processor subdomain bipartition can be computed with the help of built-in architecture definitions (algorithmically coded domain decomposition). Thus, for instance, for the hypercube target architecture, processor domains are sub-hypercubes and the corresponding processor domain bipartitioning function splits a hypercube into two sub-hypercubes. For more details on this issue, see [Pel07, PR96]. In the following, if supported by SCOTCH, the benchmarks are performed with algorithmically coded processor domain decomposition. Otherwise, a processor domain bipartitioning is computed by multilevel graph partitioning. In this case, the methods used for graph coarsening, initial bipartitioning, and refinement are, respectively, heavy edge matching, greedy graph growing, and a variant of the Fiduccia-Mattheyses method. This multilevel method is also used for mapping a process subset onto two processor subdomains.

In respect of the complexity of SCOTCH DRB, the depth of the dual recursive bipartitioning tree is $O(\log |V_n|)$. At each level of the bipartitioning tree, $O(|E_v|)$ operations are performed. Thus, the overall complexity is in $O(|E_v| \log |V_n|)$. A more detailed complexity analysis can be found in [PR96].

4.3.8 Tarun Agarwal

In [ASK06] Tarun Agarwal et al. propose a two phase non-uniform communication cost aware mapping strategy. First, in the partitioning phase, the set of processes V_v is partitioned into $|V_n|$ sets (tasks) so that heavily communicating processes are in the same set (task). This phase does not consider non-uniform network communication cost. Second, in the mapping phase, the $|V_n|$ tasks are mapped onto the $|V_n|$ processors taking non-uniform communication cost into account. The strategy's implementation in this work uses PartHom (Section 4.3.10) in the partitioning phase and, in the mapping phase, the algorithm proposed by T. Agarwal et al. The idea of their method for the mapping phase is a follows.

```
Data: V_t // Set of tasks.
      V_p // Set of processors.
      (|V_t| = |V_p| = n)

Result: M : V_t ⟶ V_p // A task mapping.

T_0 ← V_t;
P_0 ← V_p;

for (k ← 0 to n − 1) {
  /*
   * Select the next task and processor (t_k, p_k).
   * Task t_k is the one with maximal criticality.
   */
  max_criticality ← −∞;
  for (task t ∈ T_k) {
    criticality(t) = 1/(n − k) · Σ_{p∈P_k} f_est(t, p) − min_{p∈P_k} f_est(t, p);
    if (criticality(t) > max_criticality) {
      t_k ← t;
      max_criticality ← criticality(t);
    }
  }

  /*
   * Processor p_k is the one where t_k costs least.
   */
  min_cost ← ∞;
  for (processor p ∈ P_k) {
    if (f_est(t_k, p) < min_cost) {
      p_k ← p;
      min_cost ← f_est(t_k, p);
    }
  }
  M(t_k) = p_k;
  T_{k+1} ← T_k − {t_k};
  P_{k+1} ← T_k − {p_k};
}
```

Listing 4.2: T. Agarwal et al. mapping phase algorithm pseudocode ([ASK06])

During each iteration, a task is selected to be placed on the processor where, according to a *cost estimation function*, it costs least to place. This processor is called the best processor. The selection of a task is based on its *criticality*, i.e., the difference between the cost of placing a task on its best processor and the expected cost when placed on an arbitrary processor. Regarding this, the most critical task is assigned to its best processor in the current iteration. Finally, this task and its processor are marked unavailable for subsequent cycles. For the method's implementation in this work, the *second order approximation* from [ASK06] was chosen as cost estimation function (see [ASK06] for details). Listing 4.2 depicts the mapping phase algorithm. Note that T_k and P_k denote the set of tasks and the set of processors that are available at the beginning of the k^{th} iteration, respectively. Furthermore, $f_{est}(t, p)$ represents the cost estimation function and yields the cost for placing task t on processor p in the current iteration.

The running time of the implementation is determined by the complexity of the partitioning and the mapping phase which are of order $O(|E_v| \log |V_n|)$ and $O(|V_n|^3)$, respectively. Hence, the overall complexity is $O(|E_v| \log |V_n| + |V_n|^3)$.

4.3.9 Takanobu Baba

The mapping strategy by T. Baba et al. [BIY90] considers non-uniform communication cost and proceeds as follows. At each step, a process from the process graph G_v is selected and allocated to a processor of G_n. Thereby, once allocated a process is not migrated to a different processor. The most important aspect of this method are the different selection criteria for processes and processors. For process selection the criteria are:

- Number of links with already allocated processes at the k^{th} cycle
- Number of links with all the other processes
- Total distances from the other processes

Similarly, processors are selected according to the following factors:

- Total communication cost with the allocated processes
- Total distances from the other processors
- Density around a processor, i.e., how many processes have been allocated to the other processors and what are the distances

A precise description of the above properties and how these are combined to define different allocation strategies can be found in [BIY90]. Under the assumption (used for the benchmarks below) that the distances between all processes V_v in G_v and the communication cost between all pairs of processors in G_n are known in advance, the complexity of our implementation of T. Baba's mapping strategy is $O(|V_v|^2 \log |V_v|)$.

4.3.10 PartHom

The PartHom mapping algorithm is a multilevel k-way partitioning strategy based on the Chaco MLRB method. As Chaco MLRB it computes a k-way partitioning by means of recursive bisectioning. The methods used in the multilevel approach are: random matching (RM) in the coarsening phase, linear mapping (cf. Chaco Linear) for initial partitioning, and Fiduccia and Mattheyses (FM) for refinement. The complexity analysis is similar to that for Chaco MLRB. Thus, the asymptotic running time is of order $O(|E_v| \log |V_n|)$.

4.3.11 PartHet

The multilevel PartHet mapping strategy, considering non-uniform communication cost, follows the ideas of the Jostle MLkP+ method. More precisely, during the coarsening phase it uses a matching algorithm which visits the vertices in ascending order, starting with vertex 0. The intial partitioning is calculated by linear mapping (cf. Chaco Linear) and optimized by computing an approximation to the quadratic assignment problem with the help of simulated annealing (SA). Finally, in the refinement phase, an FM like algorithm similar to that used in Jostle MLkP+ is applied. However, the two refinement methods mainly differ in the load balancing algorithm. In this case, PartHet uses the Diffusion Algorithm Searching Unbalanced Domains (DASUD) method by A. Cortés et al. [CRC+02]. With reference to PartHet's complexity, see the discussion about the running time of Jostle MLkP+. In addition, it must be noted that the DASUD load balancing algorithm has been shown to be finite [CRC+02], however, to the author's knowledge, there is no proof about DASUD's speed of convergence. For these reaons, it is referred to the absolute running times of the benchmarks of PartHet to get an impression of PartHet's computational complexity.

4.3.12 Summary

Having discussed the mapping strategies which are focused on in this work, this section gives a brief summary of them. Based on the above discussion, Table 4.2 depicts each algorithm's computational complexity (Complexity), where the implementation is taken from (Implementation), and whether the mapping strategy is aware of non-uniform communication cost (Topology aware).

Mapping strategy	Topology aware	Complexity	Implementation
Chaco Linear		$O(\|V_v\|)$	Chaco
Chaco MLRB		$O(\|E_v\| \log \|V_n\|)$	Chaco
Metis MLRB		$O(\|E_v\| \log \|V_n\|)$	Metis
Metis MLkP		$O(\|E_v\|)$	Metis
Jostle MLkP		see benchmarks	Jostle
Jostle MLkP+	•	see benchmarks	Jostle
Scotch DRB	•	$O(\|E_v\| \log \|V_n\|)$	Scotch
T. Agarwal	•	$O(\|E_v\| \log \|V_n\| + \|V_n\|^3)$	by author of this work
T. Baba	•	$O(\|V_v\|^2 \log \|V_v\|)$	by author of this work
PartHom		$O(\|E_v\| \log \|V_n\|)$	by author of this work
PartHet	•	see benchmarks	by author of this work

Table 4.2: Summary of mapping strategies

Chapter 5

The Potential of Mapping Strategies

This chapter investigates to what extent and under which conditions the mapping strategies described in Chapter 4 are able to improve mapping quality over that under the linear mapping. For that reason, benchmarks were performed which compute process-to-processor mappings for a variety of process topology and network topology combinations. Finally, the results give evidence that non-trivial mapping algorithms can improve mapping quality and thus have potential for reducing communication time. The chapter is organized as follows. The first section describes the benchmarking process and the remaining two sections evaluate the results regarding mapping quality and running time of the mapping algorithms, respectively. Note that, due to the great amount of benchmark results, this chapter only presents a small subset of them. The remaining results, however, can be regarded as reference and are available in Appendix A.

5.1 The Benchmarks

Generally speaking, the benchmarks compute process-to-processor mappings for the MPI mapping problem. The input for each computation is a virtual topology (as defined by MPI), a network topology and a mapping algorithm. To model communication cost between pairs of processors in network topologies, the hop metric (Section 3.2.1) is used. According to the Scope of this Work (Section 1.2), the process and network topologies of interest are listed in Tables 5.1 and 5.2, respectively. In connection with this, regarding higher dimensional topologies, the processes/processors are distributed among the dimensions in a balanced way. That means, the sizes of the dimensions in a topology are as close as possible to each other.

Process topologies
{1,2,3}D grid
{1,2,3}D torus
Hypercube

Table 5.1: Process topologies for benchmarks

Network topologies
Fully connected network
{1,2,3}D grid
{1,2,3}D torus
Hypercube
Binary tree

Table 5.2: Network topologies for benchmarks

Furthermore, the number of processes is four times the number of processors. Thus, under a valid mapping, four processes are assigned to each processor. If a mapping does not satisfy the perfect balance requirement it is noted in the key of the corresponding graphical illustration. Finally, all mappings are evaluated in respect of their quality and the time needed by the mapping algorithm. As quality measure, the weighted edge cut Γ is preferred (see discussion on mapping cost functions in Section 3.3). However, in respect of the MPI mapping problem, for networks with non-uniform communication cost (all but fully connected network, in our case), we use Γ' (Section 3.3.2), and for networks with uniform-communication cost (fully connected network), Φ' (Section 3.3.1) is used. In order to be able to compare the performance of the different mapping strategies more easily to the linear mapping, the mapping quality results are relative to those of the linear mapping strategy implemented in Chaco Linear. Thus, values less than one denote an improvement over the linear mapping.

A closer look at the benchmark results shows that the results of some strategies are missing for larger process counts. The reason is that, due to increasing execution times for larger problem sizes and the great amount of trials performed, the corresponding computations were omitted. Due to this, results are available for running times up to ≈ 1 min. Nevertheless, the information provided should be sufficient to get an impression of the performance of the different strategies.

The last remark concerns how the benchmarks were performed. Given a virtual topology, a network topology and a mapping strategy, 10 process-to-processor mappings are computed and the average values

CPU	RAM	OS
Intel Core2 T5500, 1.66 GHz, 2 MB Cache	1 GB	Fedora 7, Linux 2.6.21

Table 5.3: Platform for the theoretical benchmarks

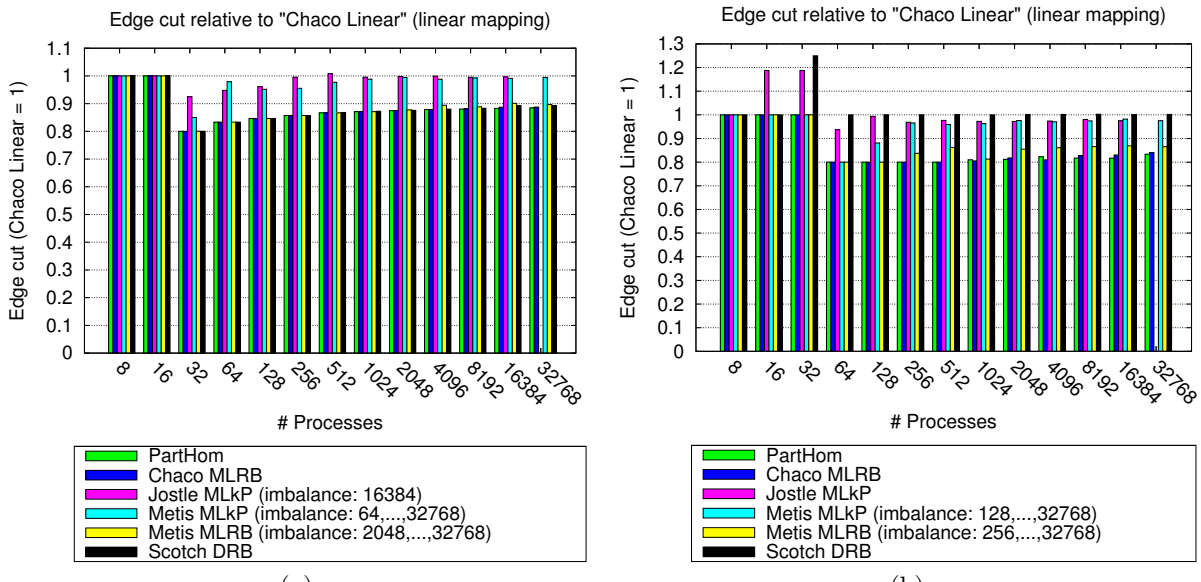
of mapping quality and running time are taken as the corresponding benchmark results. The system on which the mappings were computed is described in Table 5.3.

5.2 Mapping Quality

This section shows how mapping quality is affected by the combination of process topology and network topology and the choice of the mapping algorithm. First, networks with uniform communication cost (fully connected network) are considered. According to this, results are shown for the mapping algorithms in Chapter 4 that assume uniform network communication cost. Afterwards, the focus is on networks with non-uniform communication cost (grid, torus, etc.). In this respect, results of mapping strategies that are aware of non-uniform communication cost are presented. For an overview about the algorithms, see Table 4.2 on page 27. The absolute running times of the benchmarks presented here can be found in Appendix A.

5.2.1 Uniform Communication Cost

The first example is depicted in Figure 5.1(a). It shows the results for mapping 3D grids onto fully connected networks. As it can be seen, the strategies based on recursive bisectioning yield an improved edge cut of about 10 – 20%, compared to the linear mapping.



(a) (b)

Figure 5.1: Mapping: (a) 3D grid → fully connected network. (b) 2D torus → fully connected network.

Figure 5.1(b) depicts the results of mapping 2D tori onto fully connected networks. Here, improvements in mapping quality of about 20% are achieved with PartHom, Chaco MLRB and Metis MLRB. This performance gain does not change significantly with a growing number of processes and processors. However, the results also show that reduction in running time can only be expected for process and processor counts greater than or equal to 64 and 16, respectively. Again, the recursive bisectioning approach computes mappings that are consistently better than those under direct k-way partitioning.

5.2.2 Non-uniform Communication Cost

In Figure 5.2(a) mappings of 2D grids onto 2D grids are considered. It seems obvious that such a combination offers much room for optimization. This assumption is validated by a reduction of the weighted edge cut of about 50% for almost all strategies. Particularly, T. Agarwal's approach yields good results even for larger problem sizes in an acceptable amount of time ($\approx$ 1 min).

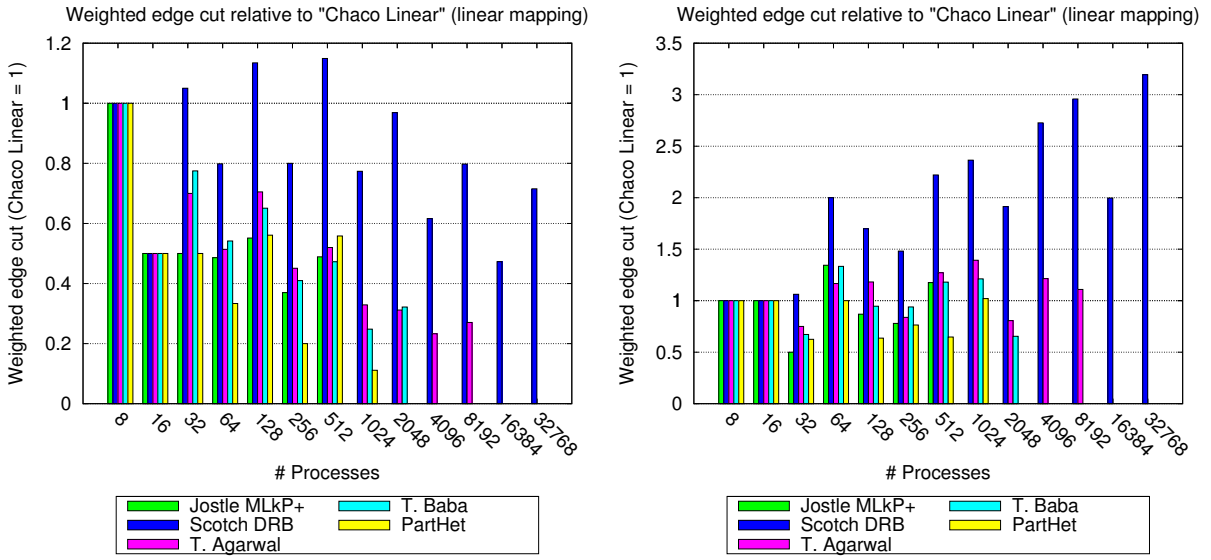


Figure 5.2: Mapping: (a) 2D grid → 2D grid. (b) 3D grid → 2D grid.

The next example (Fig. 5.2(b)) is similar to the previous one in that the same network topology is considered. However, the virtual topology is a 3D grid with an average degree of six in comparison to an average degree of four in a 2D grid. Thus, this pair of virtual and network topology has less room for optimization than the previous one. The benchmark results confirm this.

In the following benchmark (Fig. 5.3(a)) hypercubes are mapped onto 3D tori. Generally, all but two algorithms yield results that are equal to or worse than those under the linear mapping. These two, T. Agarwal and T. Baba, have similar results although they use substantially different approaches (see Sections 4.3.8 and 4.3.9). Regarding the other methods, the search space seems to be too large and they do not find the right direction.

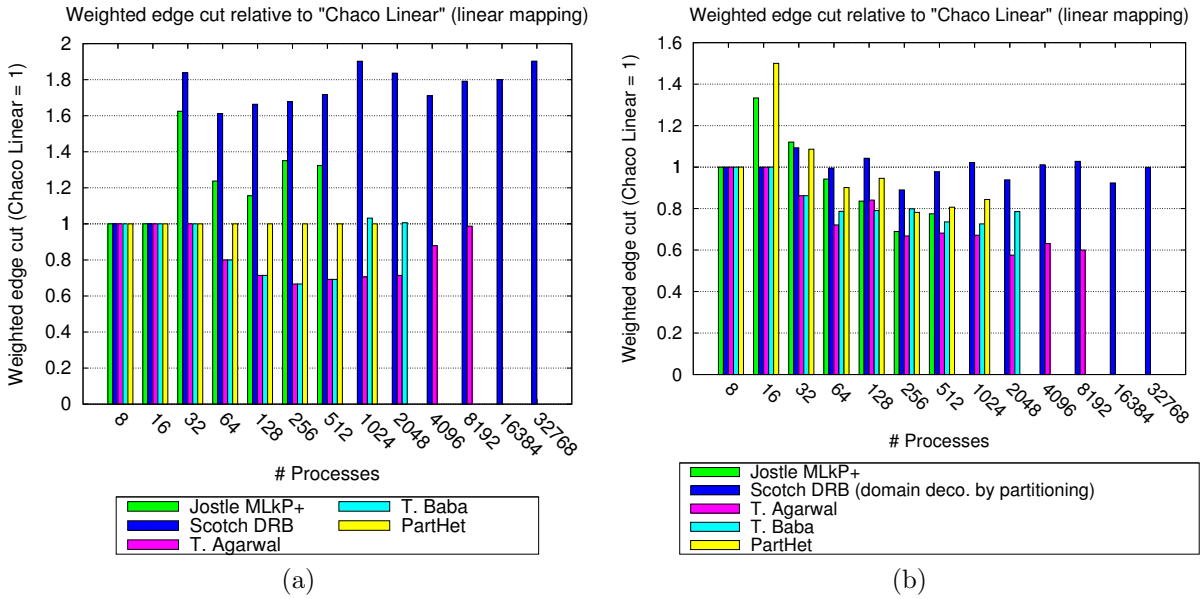


Figure 5.3: Mapping: (a) hypercube → 3D torus. (b) 3D torus → binary tree.

The last figure in this section shows the benchmark results for mapping 3D tori onto binary trees (Fig. 5.3(b)). Broadly speaking, the larger the problem size the better the results in comparison to the linear mapping. This might be due to that poor mappings, such as the linear mapping in this case, are more penalized, regarding the weighted edge cut, as the diameter and thus the maximum hop count of a topology grows. Here, the approach of T. Agarwal yields the best results.

5.3 Running Time

Having shown that non-trivial mapping strategies can improve mapping quality compared to that under the linear mapping, this section investigates another factor which is the running time of mapping strategies. This is important since optimized process-to-processor mappings are only useful if they can be computed in an acceptable amount of time. After the asymptotic running time analysis in Chapter 4, absolute running times are presented in the following. As previously mentioned in Section 5.1, the benchmarks were performed for running times of up to ≈ 1 min. Thus, this value can be regarded as upper bound on the time for computing process-to-processor mappings in this work.

5.3.1 Uniform Communication Cost

Regarding the mapping algorithms that are oblivious to network communication cost, Figure 5.4(a) shows the absolute running times for mapping 2D tori onto fully connected networks. A comparison with the remaining benchmarks for fully connected networks (see Appendix A.1) shows that the times in Figure 5.4(a) are representative of this category of benchmarks. As it can be seen, all algorithms under consideration were able to compute mappings of 32768 processes onto 8192 processors within 10 seconds. Most of them required even less time.

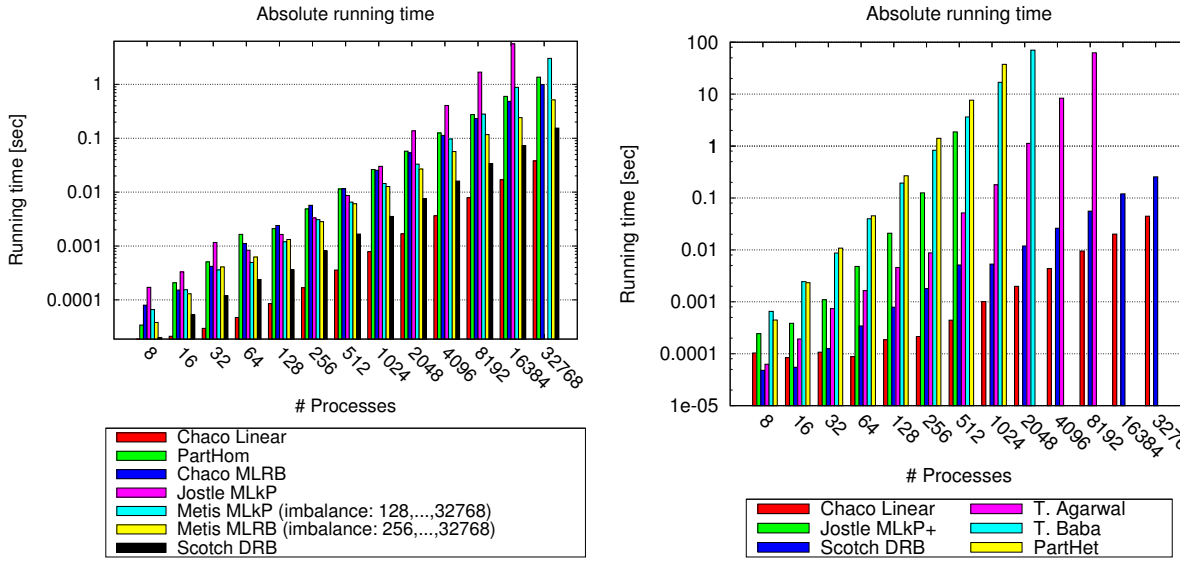


Figure 5.4: Times for mapping: (a) 2D torus → fully connected network. (b) 3D grid → 2D grid.

5.3.2 Non-uniform Communication Cost

In respect of mapping algorithms that consider variations in network communication cost, Figure 5.4(b) presents the running times for 3D grid process topologies and 2D grid network topologies. As with the remaining results for topology aware mapping strategies (see pages 47 – 78), most algorithms are able to compute process-to-processor mappings of 2048 processes onto 512 processors in less than 1.5 min.

5.4 Conclusion

This chapter demonstrated that the mapping strategies from Chapter 4 are able to improve mapping quality by up to 50%. Thereby, the time required for computing optimized process-to-processor mappings ranged from 10 sec. to 1.5 min. As consequence, the potential of mapping strategies for optimizing process-to-processor mappings in terms of the MPI mapping problem has been proven. However, these results do not necessarily imply an actual decrease in program execution time. The main reason is that, so far, we have not shown a correlation between the weighted edge cut measure and program execution time. This is done in the remainder of this work.

Chapter 6

Integration into Open MPI

In order to be able to illustrate how the quality of process-to-processor mappings affects the running time of MPI programs, this chapter discusses the integration of mapping algorithms into an existing MPI implementation in terms of the MPI process mapping facility. For this purpose Open MPI 1.2.8 was chosen. The first two sections give a short introduction to Open MPI and its architecture. Then, details about Open MPI's subsystem that is responsible for the topology mapping mechanism are presented. Finally, it is described how some of the mapping strategies from Chapter 4 have been incorporated and how they can be used. In the following, all information related to Open MPI refer to version 1.2.8.

6.1 About Open MPI

Open MPI [GFB+04] is open source and provides a full implementation of both the MPI-1 [SOHL+98] and the MPI-2 [GHLL+98] standard. It is based on ideas from prior MPI implementations and is the result of the collaboration between the authors of FT-MPI[1], LA-MPI[2] and LAM/MPI[3].

Generally speaking, Open MPI consists of three layers (from top to bottom):

- **OMPI** - Open MPI layer: Top-level MPI API and supporting logic.
- **ORTE** - Open Run-Time Environment: Run-time environment support (e.g., process control, global data store, out-of-band messaging).
- **OPAL** - Open Portability Access Layer: System portability code (e.g., high resolution timers, atomic memory operations), core support code for the component architecture, building block code (e.g., container classes) for upper layers.

All three layers make use of a component architecture, called Modular Component Architecture (MCA). A component, in this case, is similar to a software plugin. That means, it provides well-defined functionality. More details about the MCA can be found in the following section.

6.2 Modular Component Architecture

The Modular Component Architecture (MCA) is one of the key features of Open MPI. It enables third-party developers to easily add or change functionality without modifying Open MPI's source code. As consequence, the behaviour of the Open MPI library can be changed without recompilation. In essence, the MCA consists of three parts:

- **MCA base:** The core of the MCA that administers the frameworks and provides basic services to them such as finding, loading and unloading components.
- **Frameworks:** A framework can be considered as a subsystem. Each framework provides a fixed set of functionality (e.g., collective communication operations, network communicaton, process topologies) by means of its components. Every framework is different.
- **Components:** Every component belongs to exactly one framework. It provides the functionality defined by its framework. A component can be thought of as software plugin that is loaded and unloaded at runtime.

[1] http://icl.cs.utk.edu/ftmpi
[2] http://public.lanl.gov/lampi
[3] http://www.lam-mpi.org

6.3 Topology Framework

This section describes Open MPI's topology management framework (topo) which is the basis for our integration of process-to-processor mapping strategies. The topo framework is responsible for the MPI topology mechanism with its corresponding topology functions as described in Chapter 2. A component of the topo framework, called topo component, basically has to provide implementations for the following topology functions:

Cartesian topology functions
MPI_Cart_create()
MPI_Cartdim_get()
MPI_Cart_get()
MPI_Cart_rank()
MPI_Cart_coords()
MPI_Cart_shift()
MPI_Cart_sub()
MPI_Cart_map()

Graph topology functions
MPI_Graph_create()
MPI_Graphdims_get()
MPI_Graph_get()
MPI_Graph_neighbors_count()
MPI_Graph_neighbors()
MPI_Graph_map()

Table 6.1: topo framework topology functions

However, it is not necessary to implement all of them. In fact, only implementions for `MPI_Graph_map()` and `MPI_Cart_map()` need to be available. The remaining functions yields the topo framework itself.

As with other frameworks, the topo framework can have more than one component. The selection of a component is performed on a per-communicator basis. This leads to the following procedure. Once `MPI_Cart_create()` or `MPI_Graph_create()` is called by the user application, a topo component is selected for the new topology communicator. The selection is based on Open MPI run-time parameters and the priorities of the components. After the selection, the new communicator is equipped with information about its topology and gets a pointer to a structure of function pointers to the backend functions that provide the functionality to the high-level MPI functions from Table 6.1. This structure is called *module* in the MCA terminology.

6.4 Two New Topology Components

To incorporate our mapping algorithms into the topo framework, two topo components have been developed. The first one assumes uniform network communication cost and is called una, which stands for *uniform network access*. The second one, nuna (*non-uniform network access*), considers non-uniform network communication cost. Since the focus is on cartesian process topologies, both components implement an optimized `MPI_Cart_create()` function. In particular, if an MPI program calls `MPI_Cart_create()` with `reorder = 1` an optimized process-to-processor mapping is computed by the process with rank `0`. Afterwards, the new mapping is broadcast to the remaining processes which reorder process ranks according to the new mapping and finish the creation of the new topology communicator.

Due to the design of the topo framework, the two components only work properly if all processes in `MPI_COMM_WORLD` belong to the new topology communicator. The reason is that the backend function of `MPI_Cart_create()` does not get the old communicator as input and the new one is not ready to be used for communication, yet. Thus, there is no communicator available except `MPI_COMM_WORLD`. One possible workaround is to use out-of-band[4] communication. However, adapting the topo framework would be more appropriate.

6.4.1 The una Component

The una component assumes uniform network communication cost and uses the PartHom (Section 4.3.10) mapping algorithm for computing optimized process-to-processor mappings. PartHom was chosen because of its good performance regarding both mapping quality and running time (see Appendix A.1 for benchmark results).

The una component can also be regarded as framework for other mapping algorithms which are oblivious to network communication cost. That means, additional mapping strategies can be easily incorporated. The corresponding interface is simple, i.e., a mapping algorithm has to take the following parameters:

[4] Communication outside of MPI channels

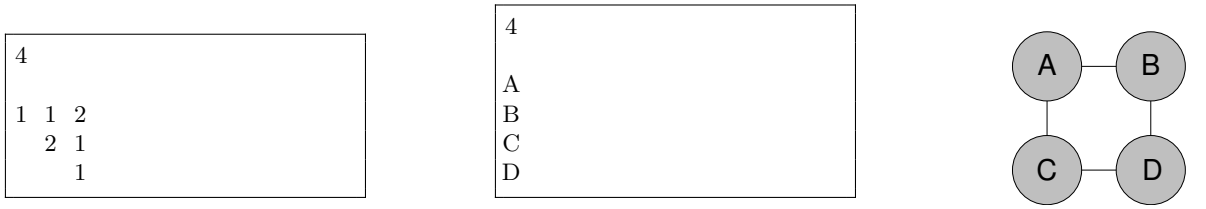


Figure 6.1: Example configuration for the nuna component

Input parameters:

- Number of processes in the process topology
- Graph representation of the process topology
- Number of processors (hosts) onto which the processes are to be mapped
- Size of each processor (number of processes for each host)

Output parameters:

- Process-to-processor mapping

6.4.2 The nuna Component

The nuna component has been developed for networks with non-uniform communication cost. Currently, it provides the following network topology aware process-to-processor mapping strategies:

- Tarun Agarwal (Section 4.3.8)
- Takanobu Baba (Section 4.3.9). Note that the implementation of this algorithm in the nuna component has, in contrast to the one with complexity $O(|V_v|^2 \log |V_v|)$ in the theoretical benchmarks (Appendix A), complexity $O(|V_v|^3)$.
- PartHet (Section 4.3.11)

In contrast to the una component, the nuna component takes three parameters.

- `topo_nuna_ncm_path`
  The path where to look for the network cost matrix. The default value is `~/.openmpi/ncm`. The network cost matrix is symmetric and contains the cost to communicate for all pairs of processors. For a discussion about the choice of coefficients for this matrix, see Sections 3.1 and 3.2. The corresponding file has two entries. First, the number of processors (hosts) that execute the MPI program. Second, the upper-triangular network cost matrix. An example is depicted in Figure 6.1, where the hop metric (Section 3.2.1) is used as communication cost measure.
- `topo_nuna_ncm_h2i_path`
  The path where to look for the mapping between hostname and network cost matrix index. The default is `~/.openmpi/ncm_h2i`. Without additional information it is not clear which coefficient in the network cost matrix stands for which pair of processors (hosts). For that reason, the file `~/.openmpi/ncm_h2i` is introduced. It has two entries. The first one is the number of processors (hosts) that execute the MPI program. The second one lists the hostnames of the hosts that run the MPI program. Thereby, the order of the hostnames corresponds to the order of indices in the network cost matrix. In other words, if numbering starts from one, the network cost matrix entry $(1, 2)$ is that for the communication cost between the first and the second host in the list of hostnames. See the example in Figure 6.1.
- `topo_nuna_map_strat`
  Which mapping strategy to use:
  `0` : Tarun Agarwal
  `1` : Takanobu Baba

`2` : PartHet

The default is `0` (Tarun Agarwal).

Similarly to the `una` component, the `nuna` component can serve as framework for other process-to-processor mapping strategies that are aware of non-uniform network communication cost. The interface is defined by the parameters below:

Input parameters:

- Number of processes in the process topology
- Graph representation of the process topology
- Number of processors (hosts) onto which the processes are to be mapped
- Size of each processor (number of processes for each host)
- Network cost matrix

Output parameters:

- Process-to-processor mapping

Chapter 7

Practical Results

Having implemented an optimized process mapping facility in Open MPI as described in Chapter 6, the purpose of this chapter is twofold. In particular, it is investigated how optimized process-to-processor mappings affect actual program execution time and to what extent the weighted edge cut, which is used as measure for mapping quality (Chapter 5), correlates with the time spent in communication. For these purposes, benchmarks have been performed for the mapping strategies that were incorporated into Open MPI. The first section describes the benchmarks and the second one discusses the results.

7.1 The Benchmarks

Broadly speaking, two kinds of benchmarks were performed. The first type considers uniform network communication cost and the second type accounts for non-uniform communication cost in the interconnect. Both run the same MPI program which creates a cartesian process topology, with request for process reordering, and all processes in this virtual topology exchange data with their nearest neighbours (see Listing 7.1). The process topologies under consideration are available in Table 7.1. Note that, for higher dimensional topologies, the processes are distributed among the dimensions in a balanced way.

Process topologies
{1,2,3}D grid
{1,2,3}D torus
Hypercube

Table 7.1: Process topologies for practical benchmarks

During the benchmark process, the execution time of the MPI program is measured for different message sizes. Furthermore, the mapping quality relative to the linear mapping (no process rank reordering) is computed. Per message size, each process exchanges 5000 messages with each of its neighbours. This is performed 25 times. Finally, the average running time of the 25 runs is taken as running time result. The underlying parallel machine is the *CHiC*[1] system at Chemnitz University of Technology. In the following, the benchmark environments for the two benchmark types are described.

```
/* Data exchange along each dimension */
for (dim = 0; dim < ndims; dim++) {
    MPI_Cart_shift(comm_cart, dim, 1, &rank_pred, &rank_succ);

    MPI_Sendrecv(sbuffer, size, MPI_CHAR, rank_succ, 1,
                 rbuffer, size, MPI_CHAR, rank_pred, 1,
                 comm_cart, &status);

    MPI_Sendrecv(sbuffer, size, MPI_CHAR, rank_pred, 1,
                 rbuffer, size, MPI_CHAR, rank_succ, 1,
                 comm_cart, &status);
}
```

Listing 7.1: Data exchange with nearest neighbours

[1] http://www.tu-chemnitz.de/chic

CPU	2 × Dual-Core AMD Opteron 2218, 2.6 GHz, 2 MB Cache
RAM	4 GB
NIC	Mellanox Technologies MT25204
OS	Scientific Linux 4.4, Linux 2.6.9

Table 7.2: CHiC compute node

7.1.1 Uniform Communication Cost

These benchmarks are performed on 12 nodes (see Table 7.2 for node description) which are connected via Infiniband to one crossbar switch. Hence, communication cost between nodes can be regarded as uniform. Each of the nodes executes 4 MPI processes. Thus, except for the hypercube process topology, the total number of processes in one job is 48. Due to the construction of the hypercube, 8 nodes (instead of 12) with 4 processes each are used. The Open MPI `una` component (Section 6.4.1) is chosen for the topology mechanism. The version of the Open MPI implementation is 1.2.4.

7.1.2 Non-uniform Communication Cost

The benchmarks are performed on 24 nodes (see Table 7.2 for node description) which are connected to two Inifiniband crossbar switches (12 per switch). The two switches are leaves in a 2-stage fat tree network which is built of 44 leaf switches (24 ports each) and two top switches (288 ports each). Hence, the communication cost between all pairs of processors cannot be considered uniform. This is only true for the nodes connected to the same crossbar switch. Each of the nodes executes 4 MPI processes. Hence, except for the hypercube process topology, 96 processes constitute one MPI job. For the hypercube, 16 nodes (12 on the first and 4 on the second switch) with 4 processes each are used. Here, Open MPI 1.2.4 and the `nuna` component (Section 6.4.2) are utilized. The coefficients for the `nuna` component's network cost matrix are chosen as follows: Entries for pairs of nodes connected to the same switch are set to 1. Pairs of nodes on different switches get the value 2.

7.2 Results

This section discusses the results of the benchmarks regarding reduction in program execution time by means of optimized process-to-processor mappings, and the correlation between the weighted edge cut mapping quality measure and program execution time. First, the results for uniform communication cost are considered. Afterwards, the focus is on non-uniform communication cost.

7.2.1 Uniform Communication Cost

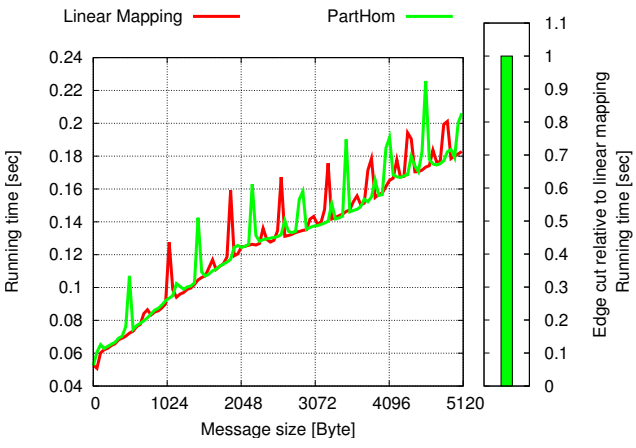


Figure 7.1: 1D grid process topology

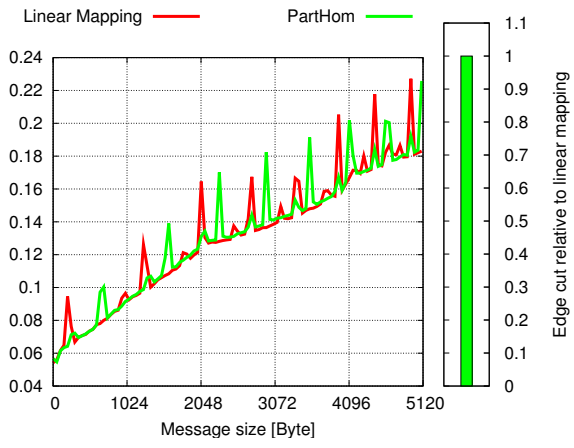


Figure 7.2: 1D torus process topology

The first three examples show that it is not always possible to reduce communication time by sophisticated process-to-processor mapping strategies. This is the case, especially if the default mapping is already an optimal mapping. Figures 7.1, 7.2 and 7.7 (on page 40) illustrate this for the 1D grid, 1D torus and hypercube process topology, respectively. As it can be seen, the running times under the linear mapping are essentially identical to that under a mapping computed by the PartHom algorithm. This is also indicated by equal edge cut values in all three cases.

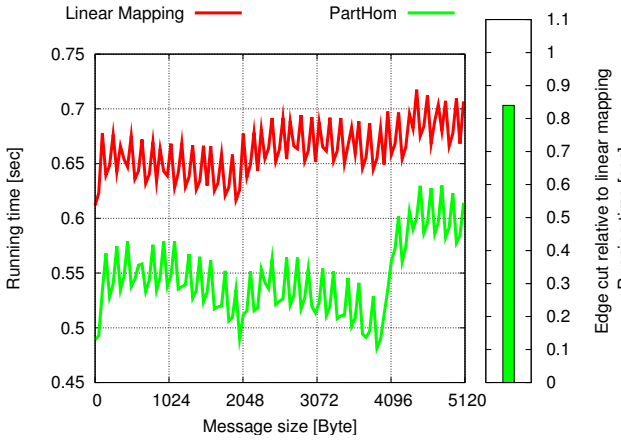


Figure 7.3: 8×6 2D grid process topology

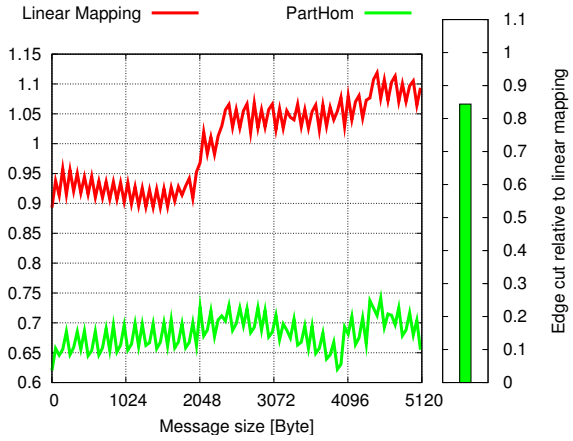


Figure 7.4: 8×6 2D torus process topology

The remaining results, however, prove that the linear mapping is not always the best choice. In particular, the edge cut could be improved by about 15% for 2D grid and 2D torus process topologies (see Figures 7.3 and 7.4). However, this does not necessarily correspond to 15% gain in actual program running time, as shown for Figure 7.4 with a reduction of about 36%.

In the last two Figures (7.5 and 7.6) another property of the edge cut is demonstrated. Specifically, both examples show gains in edge cut and running time. A closer look, however, reveals that the improvements for the 3D grid process topology, compared to the 3D torus topology, are greater regarding both edge cut and running time. Thus, although not completely correlated with program execution time, the amount by which the edge cut is improved indicates to what extent improvements in program running time can be expected. Nevertheless, it must be noted that this also depends on the characteristics of parallel programs. For instance, compute bound programs, i.e. those whose running time is determined by computation, do not benefit from reduction in communication time to the same extent as communication bound applications (running time is determined by communication) do. Our benchmarks are communication bound.

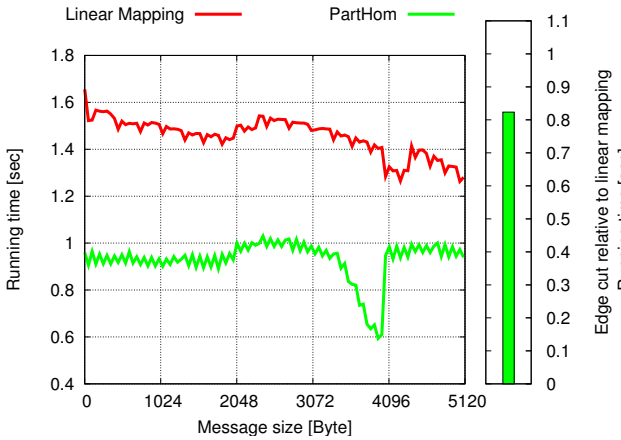


Figure 7.5: $4 \times 4 \times 3$ 3D grid process topology

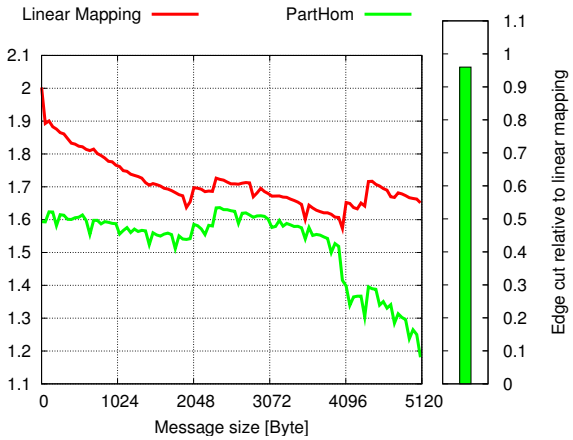


Figure 7.6: $4 \times 4 \times 3$ 3D torus process topology

7.2.2 Non-uniform Communication Cost

In order to discuss the performance of uniform communication cost mapping algorithms in connection with networks with non-uniform cost, the PartHom algorithm is benchmarked in this section, additionally.

Similarly to the previous section, the Figures 7.9, 7.10 and 7.8 show the cases where the linear mapping yields an optimal process-to-processor mapping and thus no improvement in both weighted edge cut and running time is possible.

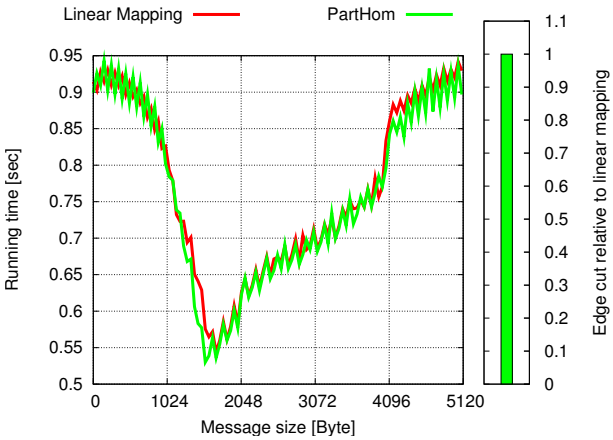


Figure 7.7: 5D hypercube process topology

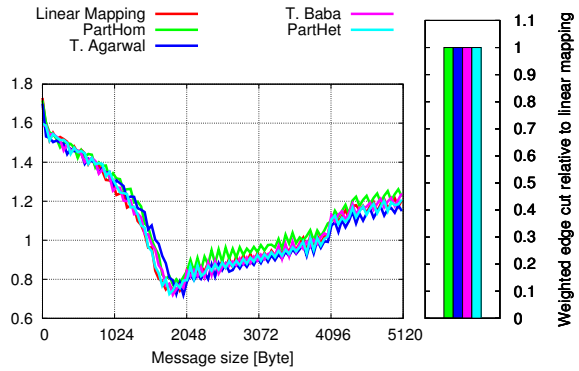


Figure 7.8: 6D hypercube process topology

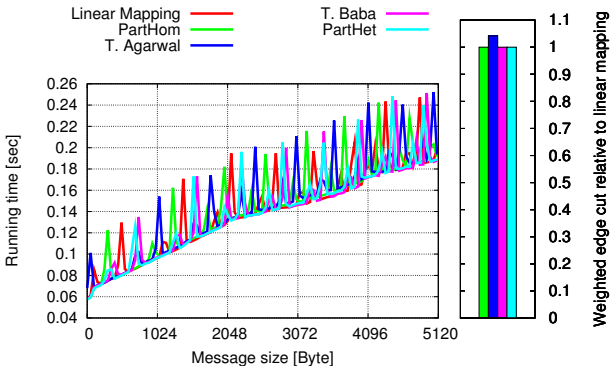


Figure 7.9: 1D grid process topology

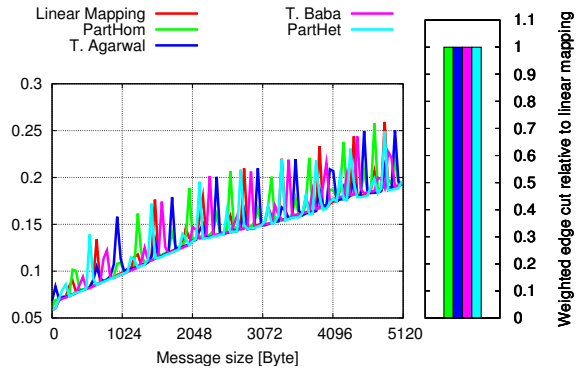


Figure 7.10: 1D torus process topology

The next two examples (Figures 7.11 and 7.12) illustrate the results for 2D grid and 3D grid process topologies, respectively. For the former, gains in running time of about 60% are achieved for small message sizes ($\approx$ 500 Byte). However, this gain decreases as message sizes increase. It is also important to note that the weighted edge cut values for PartHom and T. Agarwal suggest an improvement in running time, however, the opposite is the case. Thus, in this example, the weighted edge cut is not a reliable indicator of actual communication time.

Regarding 3D grid process topologies, a reduction in running time of about 40% is shown, compared to the linear mapping. It is also remarkable that the times for the PartHom method are similar to those under the T. Agarwal algorithm, which is also true for 2D grid process topologies (Figure 7.11). The reason is that the T. Agarwal method uses PartHom for its partitioning phase (Section 4.3.8). However, for message sizes greater than 4 KB, the results for T. Agarwal are better than those for PartHom. This is due to that T. Agarwal's mapping phase considers non-uniform network communication cost.

Figure 7.13 shows the benchmark results for 2D torus virtual topologies. Although the weighted edge cut could be improved, no performance gain could be achieved in terms of program execution time. In addition, it can be seen that the running times for all mapping strategies fluctuate very much. The reason for this, however, is not clear and requires a more in-depth analysis.

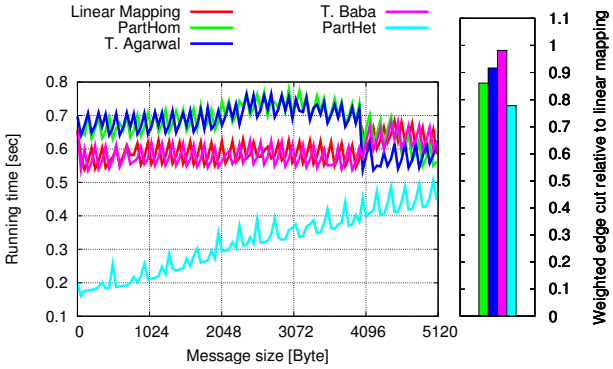


Figure 7.11: 12×8 2D grid process topology

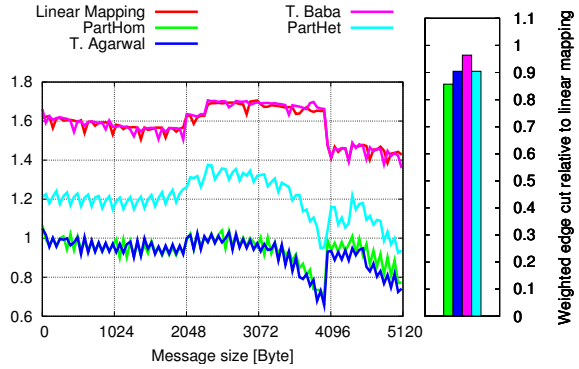


Figure 7.12: $6 \times 4 \times 4$ 3D grid process topology

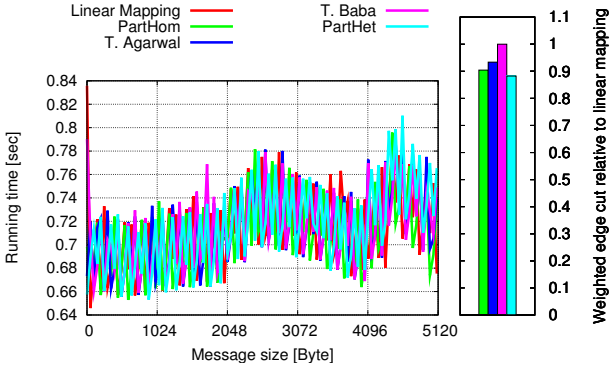


Figure 7.13: 12×8 2D torus process topology

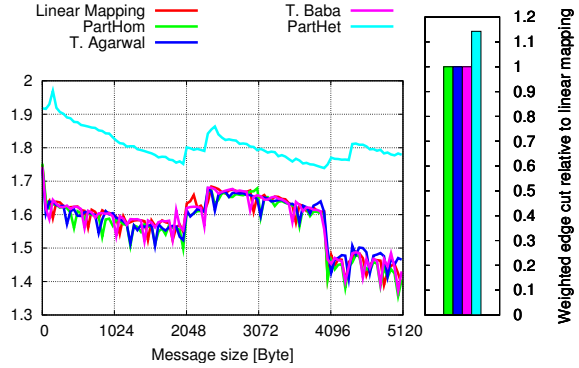


Figure 7.14: $6 \times 4 \times 4$ 3D torus process topology

Finally, the last example (Figure 7.14) demonstrates how mapping quality (weighted edge cut) can become even worse with non-trivial process-to-processor mapping algorithms. In particular, the PartHet method yields a mapping whose weighted edge cut is about 10% worse compared to that of the linear mapping. As consequence, the running time increases by 24%.

7.2.3 Conclusion

The practical results have demonstrated that program execution time can benefit from optimized process-to-processor mappings. This was shown for both uniform and non-uniform network communication cost. However, the benchmarks also give evidence that there is not a single mapping strategy which is superior to all the others. In contrast, for each mapping algorithm exist cases where its mappings improve running time and cases in which no performance gain can be achieved. Sometimes, compared to the linear mapping, even worse results are possible. It was also shown that, in general, the weighted edge cut is appropriate to assess mapping quality in terms of communication time. Thus, a relation between weighted edge cut and program execution time was found. Finally, the examples also showed that considering non-uniformity in network communication cost can yield better results in comparison to simply ignoring it (discussion about PartHom for non-uniform communication cost).

Chapter 8

Conclusion

8.1 Conclusion and Future Work

This work dealt with the MPI process topology mechanism and how it can be used to reduce MPI program execution time regarding regular process and regular network topologies. For this purpose, the MPI topology functions were discussed and the problem of finding an optimal assignment of processes to processors was formalized as the mapping problem, which cannot exactly be solved in polynomial time.

Based on the mapping problem formalization, it was investigated how to assess mapping quality in connection with process-to-processor mappings. In this respect, the discussion revealed that good mappings favour nearest neighbour processor communication. For that reason, the weighted edge cut function was chosen as measure for mapping quality. Since the weighted edge cut takes non-uniformity in network communication cost into account, different approaches for estimating communication cost between processors were considered. Examples are the hop metric and various parallel computational models (LogP, LogGP, HLogGP, LoGPC) which use different information for predicting communication time. However, none of them was regarded as the best one. The reason is that the choice should depend on the mapping strategy since the aim of modelling communication cost between processors is to *guide* a mapping strategy in assigning communicating processes to nearby processors, and every strategy behaves differently.

In the next step, state-of-the-art mapping strategies were discussed with respect to how they work and asymptotic complexity. To show the potential of these algorithms for improving mapping quality, benchmarks were performed in which gains (compared to the linear mapping) of about 20%, for uniform network communication cost, and 50%, for non-uniform network communication cost, could be achieved.

In order to examine how the gains in mapping quality affect actual program execution time, some mapping strategies were incorporated into the topology framework of Open MPI and practical benchmarks were performed on the *CHiC* cluster. Here, it could be shown that the weighted edge cut is related to communication time. Furthermore, improvements in program execution time of up to about 36% (uniform network communication cost) and 60% (non-uniform network communication cost) were obtained.

In respect of future work, a modification of the Open MPI `topo` framework is intended so that the `una` and `nuna` components can be used without restrictions.

8.2 Acknowledgements

I would like to thank Torsten Mehlan for always finding the time for discussions regarding this work. I would also like to thank Prof. Rehm. Without him this work would not have been possible. Finally, my thank goes to my parents who supported me not only during writing this diploma thesis, but during all my studies. Thank you very much.

Appendix A

Theoretical Benchmarks

A.1 Fully Connected Network

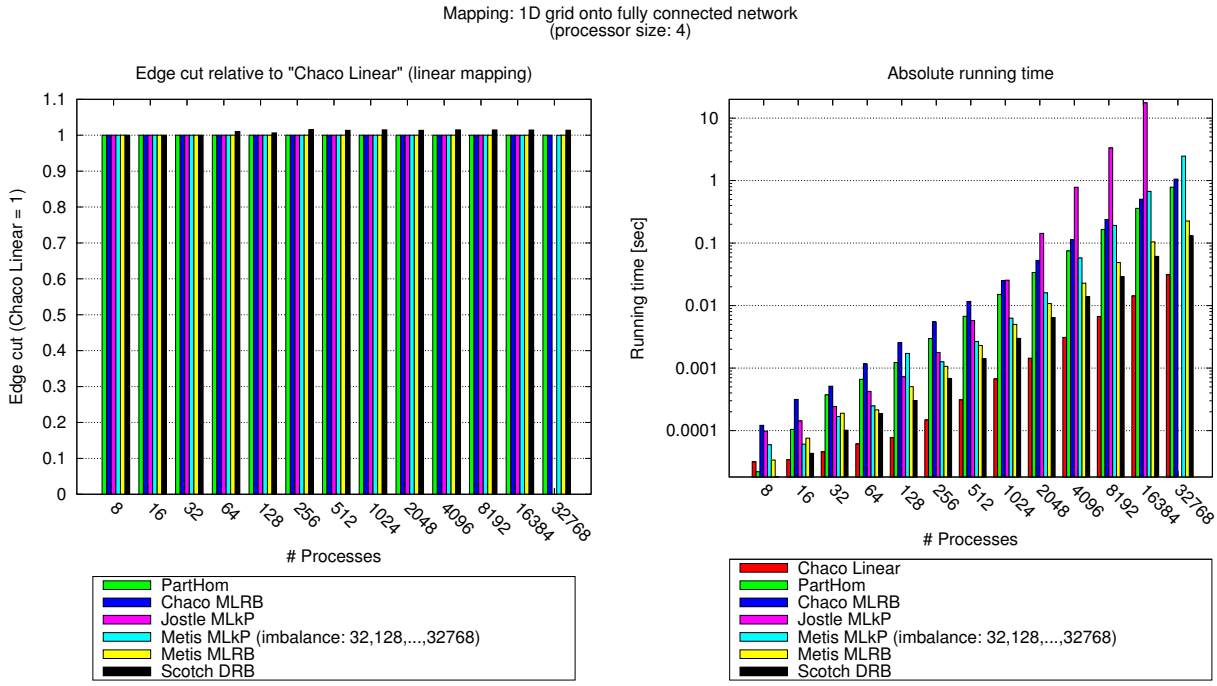


Figure A.1: Uniform communication cost mapping: 1D grid onto fully connected network

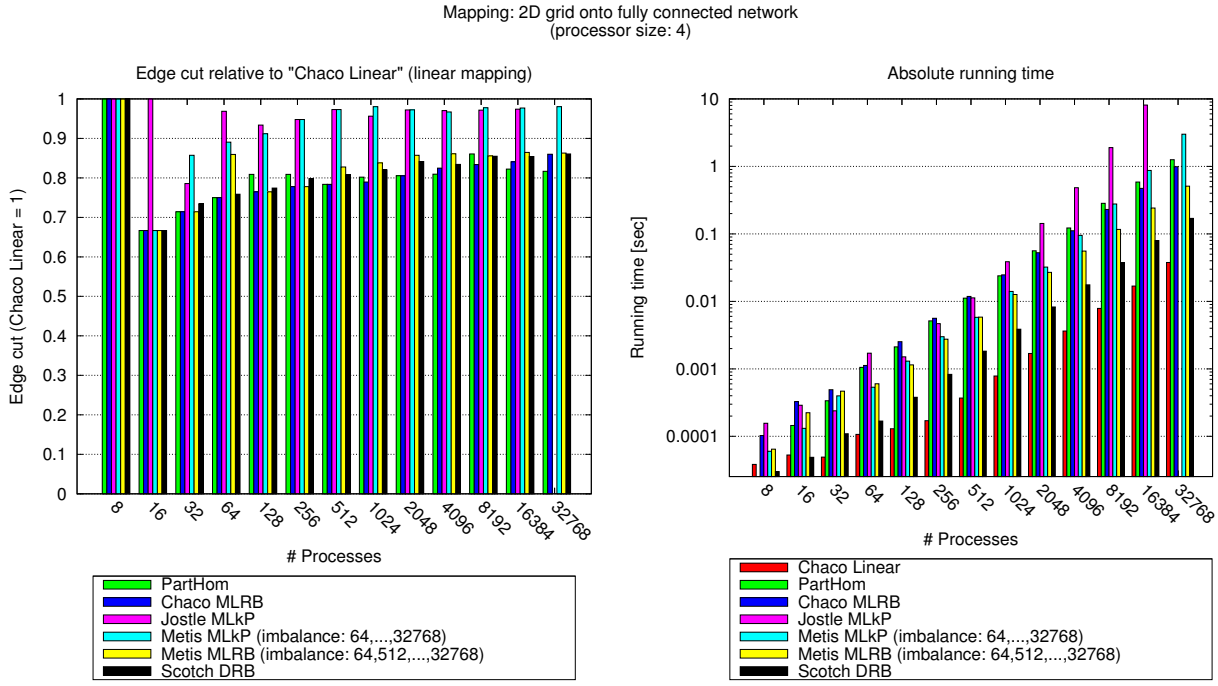


Figure A.2: Uniform communication cost mapping: 2D grid onto fully connected network

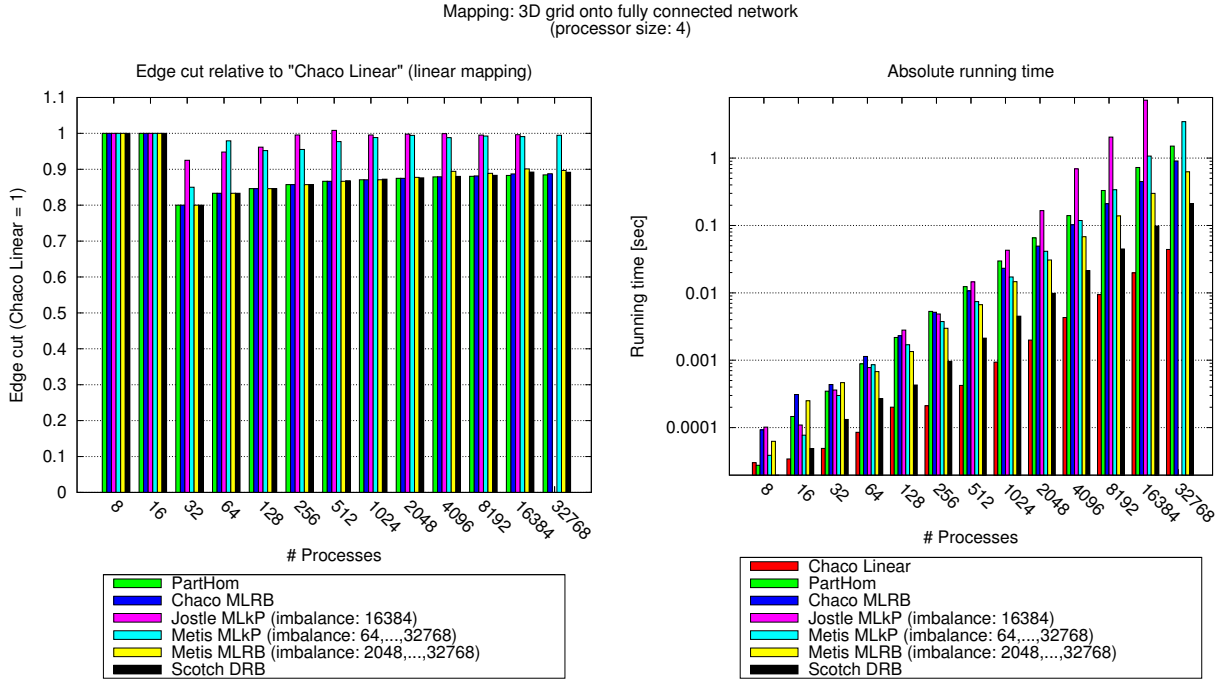


Figure A.3: Uniform communication cost mapping: 3D grid onto fully connected network

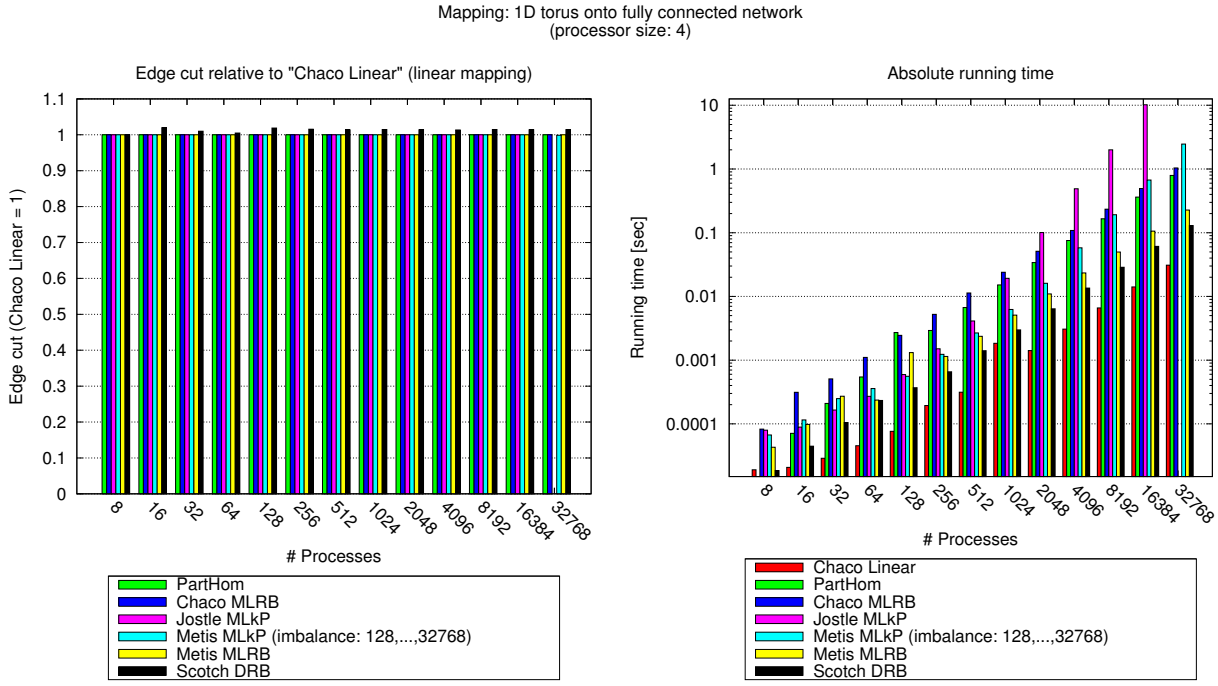


Figure A.4: Uniform communication cost mapping: 1D torus onto fully connected network

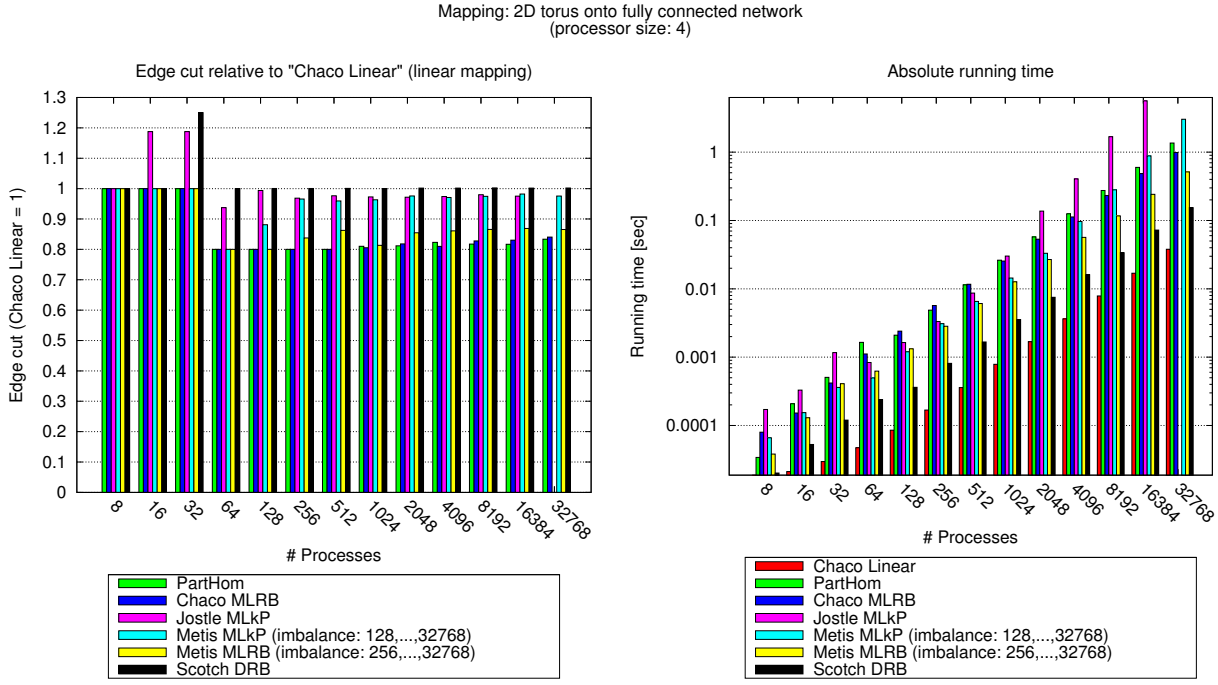


Figure A.5: Uniform communication cost mapping: 2D torus onto fully connected network

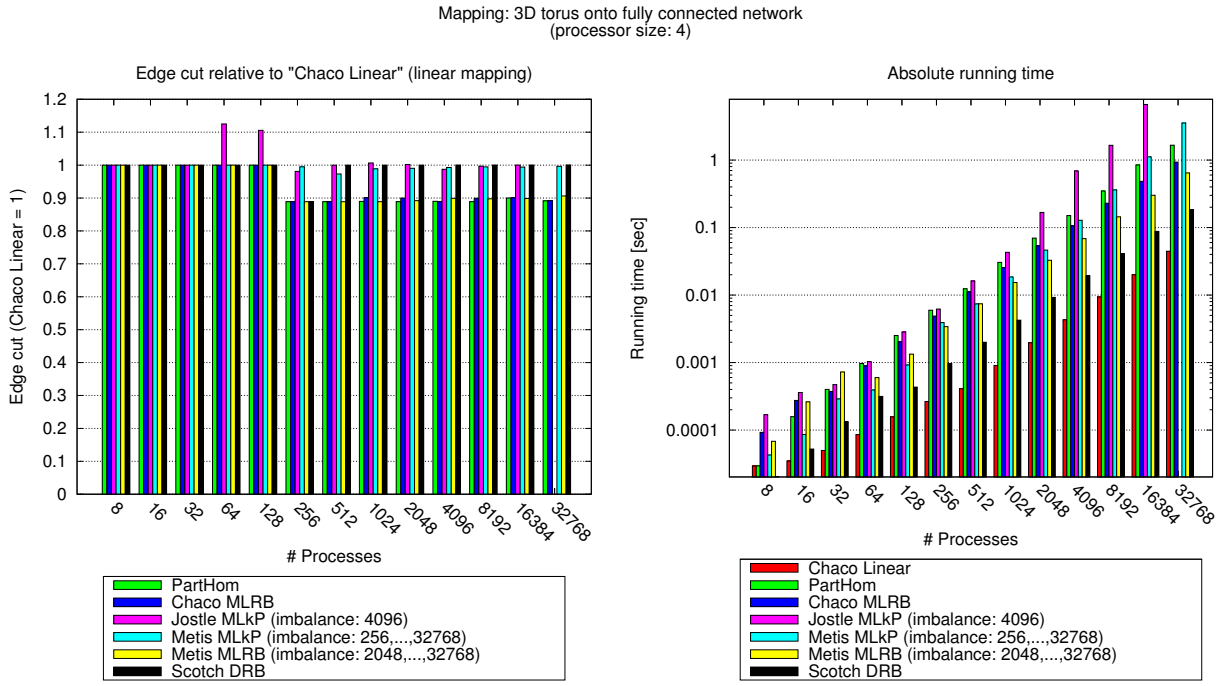


Figure A.6: Uniform communication cost mapping: 3D torus onto fully connected network

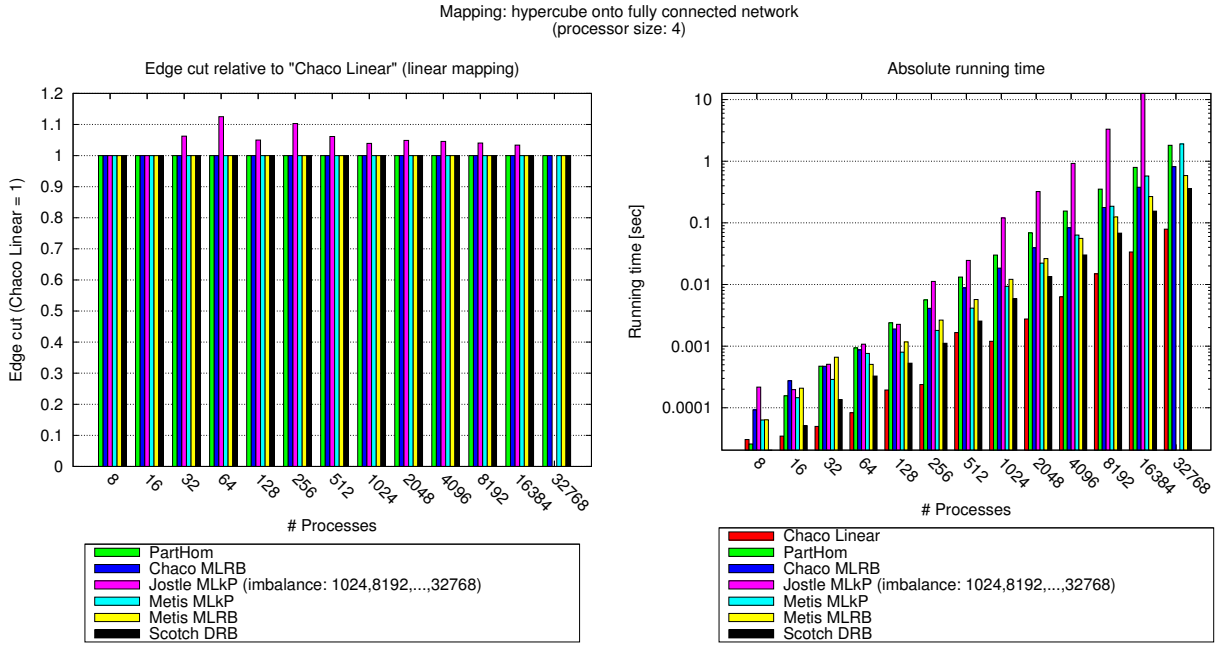


Figure A.7: Uniform communication cost mapping: hypercube onto fully connected network

A.2 1D Grid Network Topology

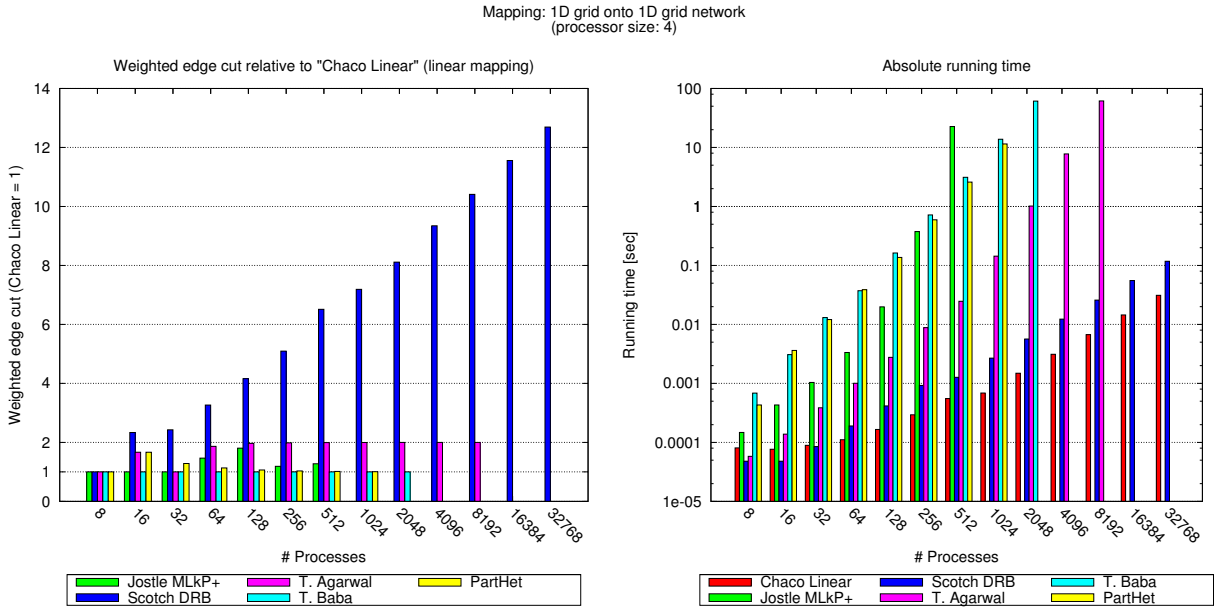


Figure A.8: Non-uniform communication cost mapping: 1D grid onto 1D grid network

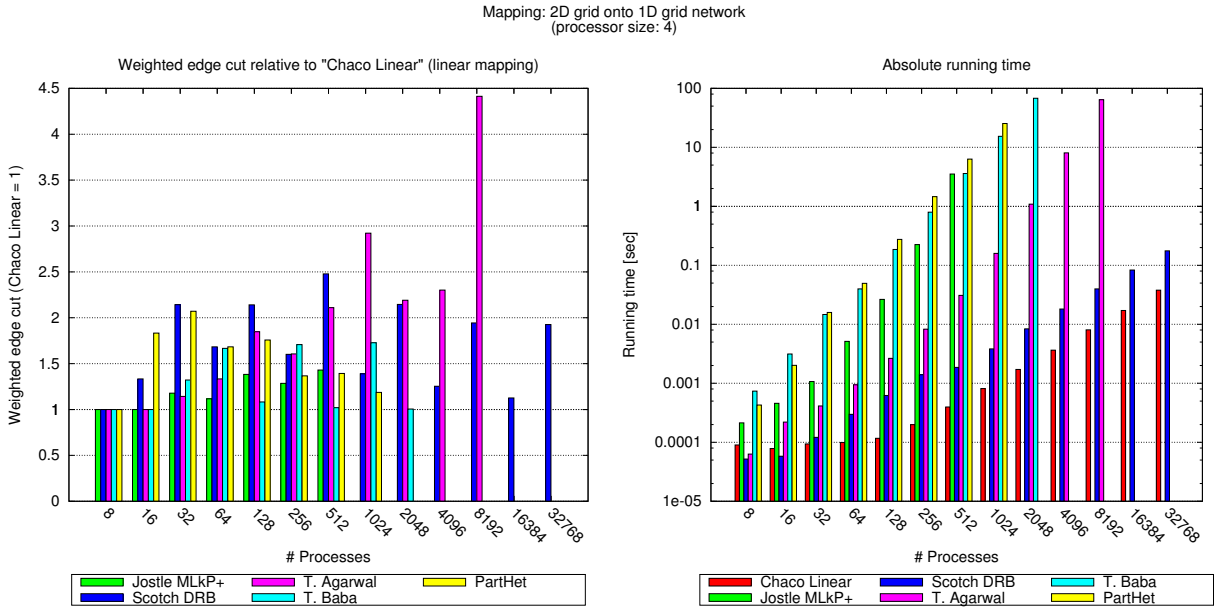


Figure A.9: Non-uniform communication cost mapping: 2D grid onto 1D grid network

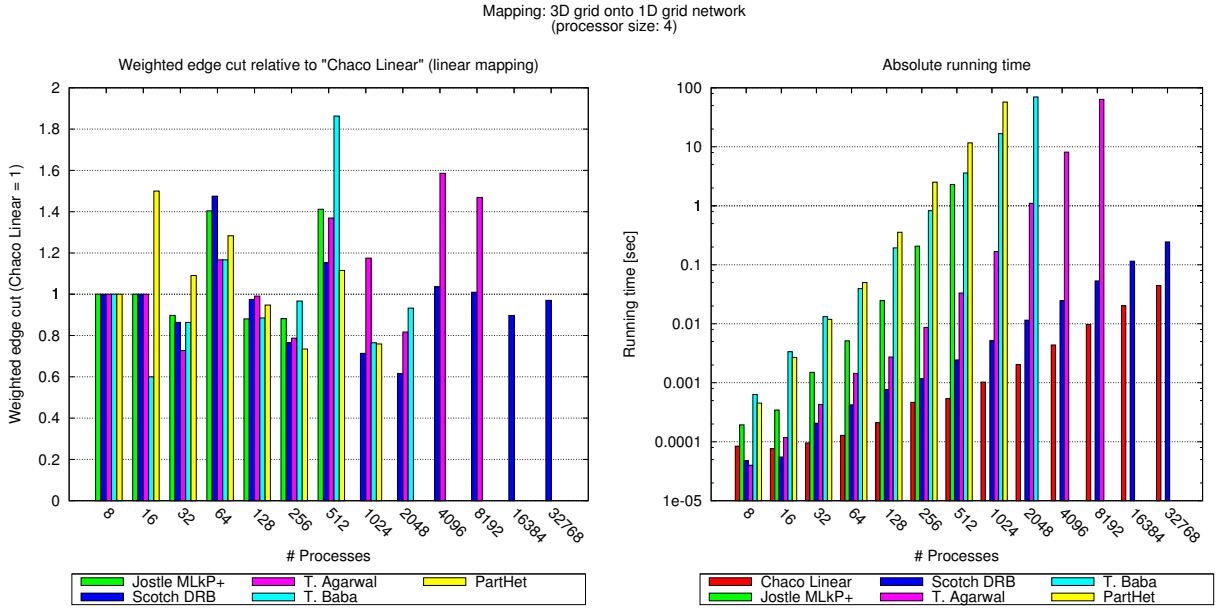


Figure A.10: Non-uniform communication cost mapping: 3D grid onto 1D grid network

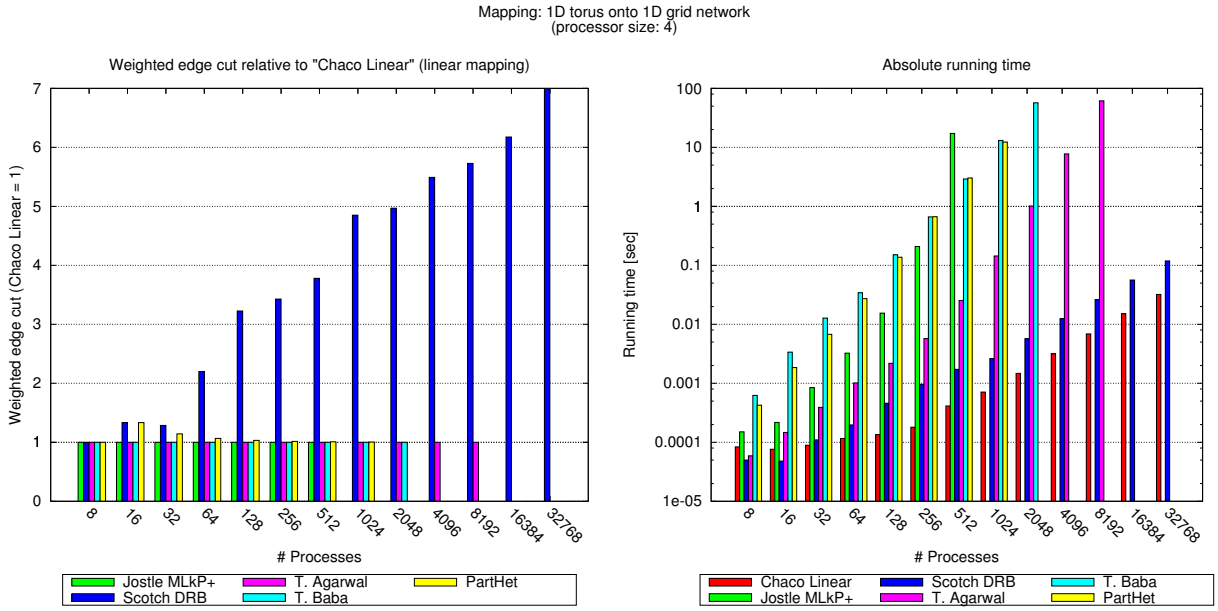


Figure A.11: Non-uniform communication cost mapping: 1D torus onto 1D grid network

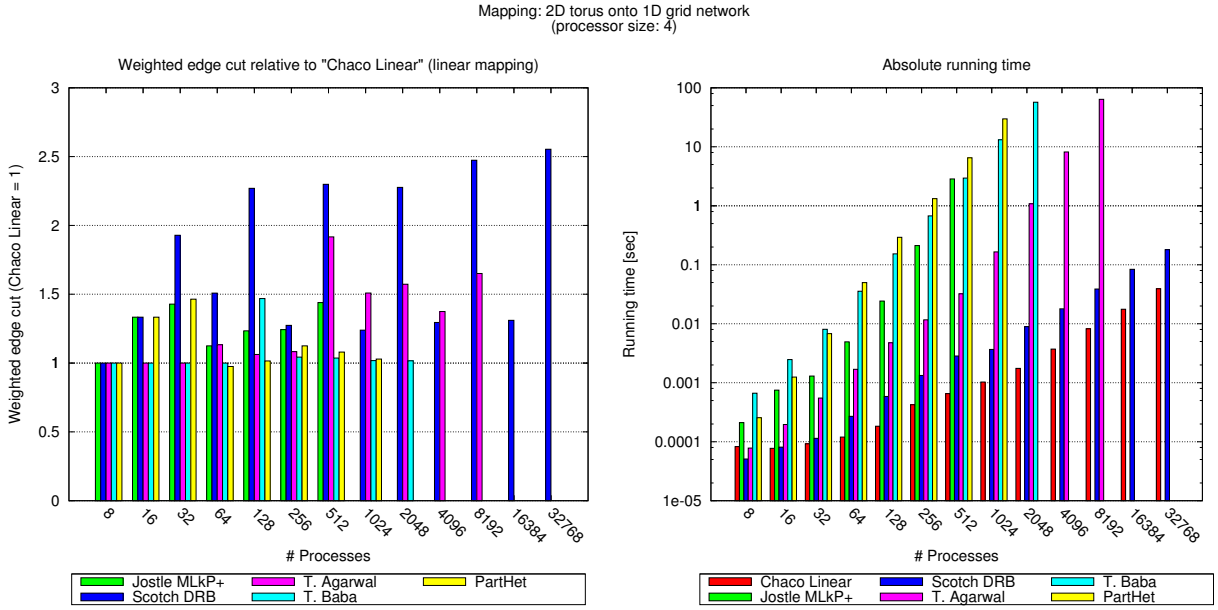


Figure A.12: Non-uniform communication cost mapping: 2D torus onto 1D grid network

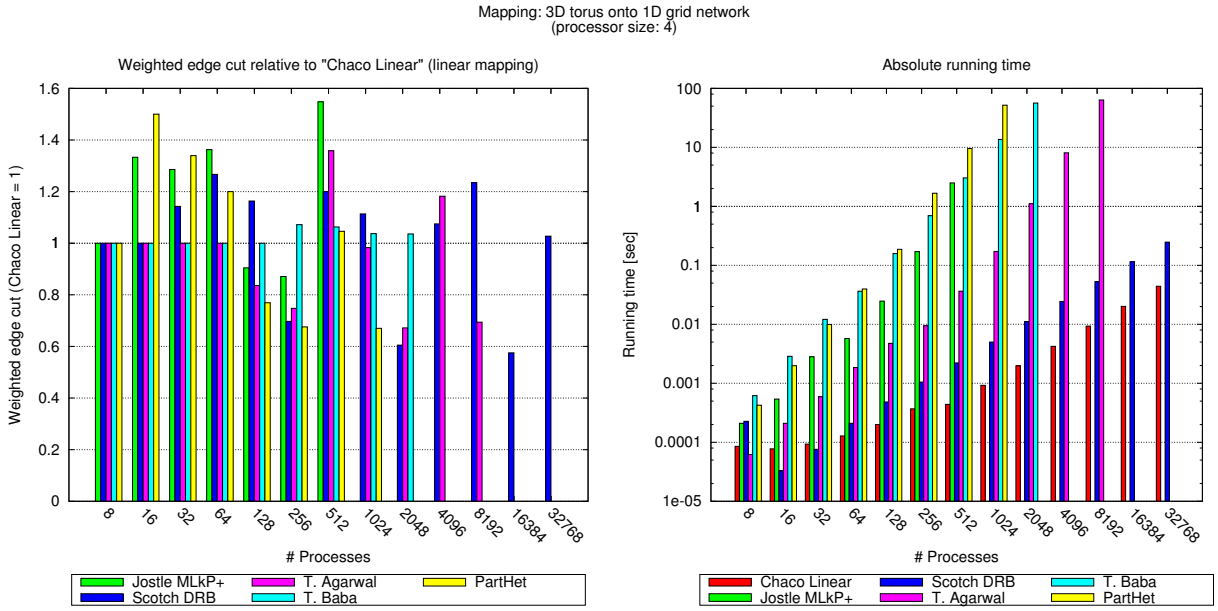


Figure A.13: Non-uniform communication cost mapping: 3D torus onto 1D grid network

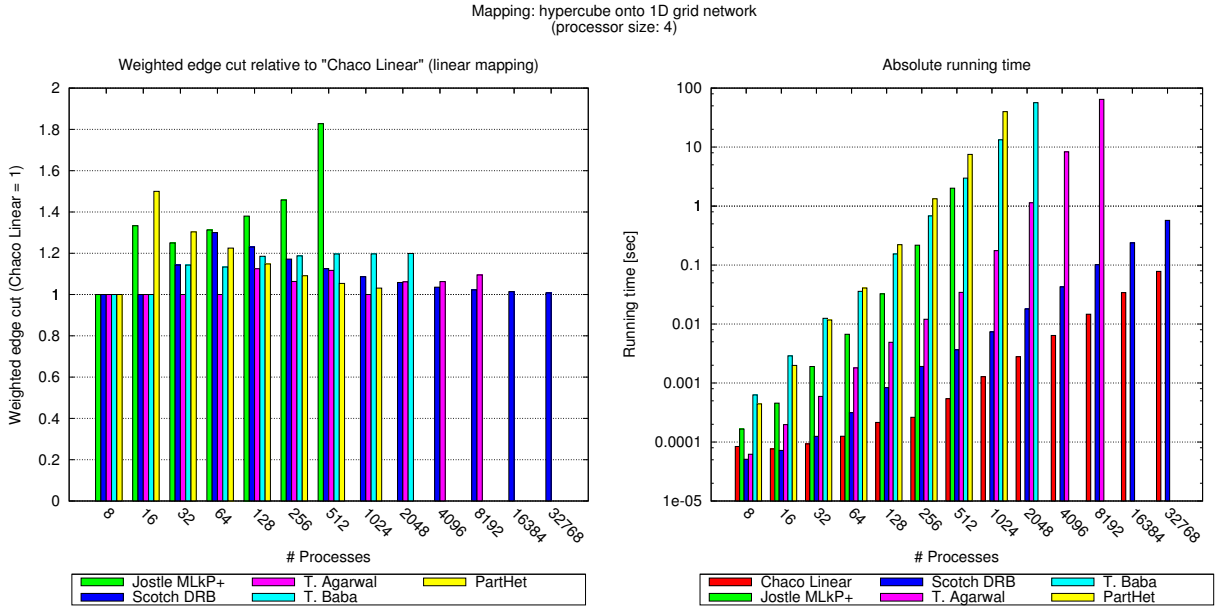


Figure A.14: Non-uniform communication cost mapping: hypercube onto 1D grid network

A.3 2D Grid Network Topology

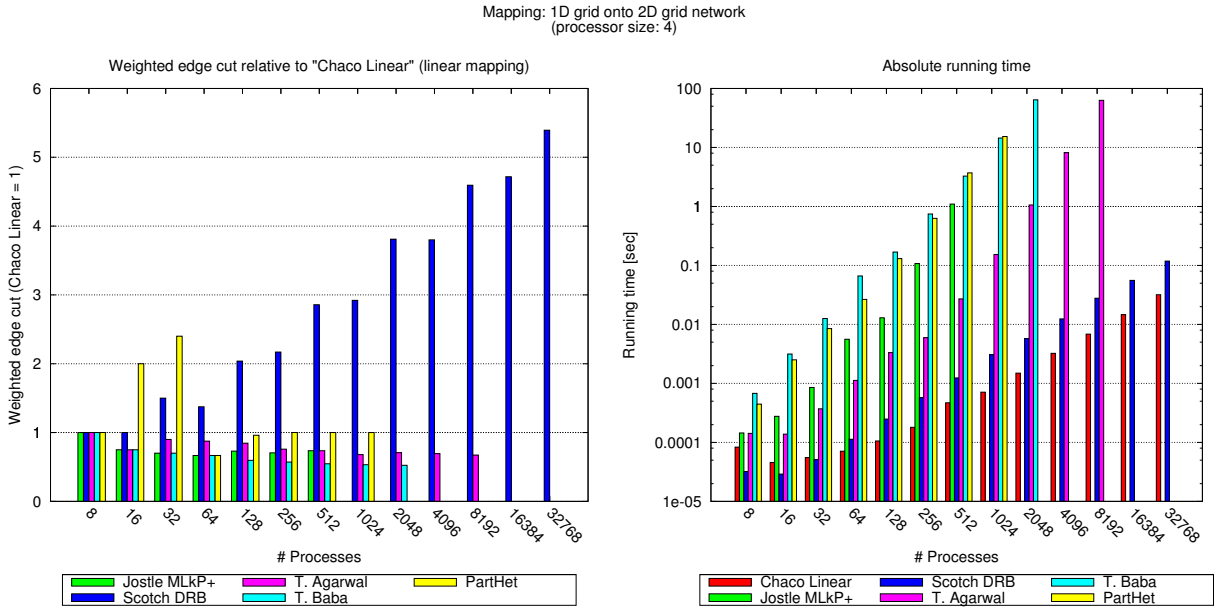


Figure A.15: Non-uniform communication cost mapping: 1D grid onto 2D grid network

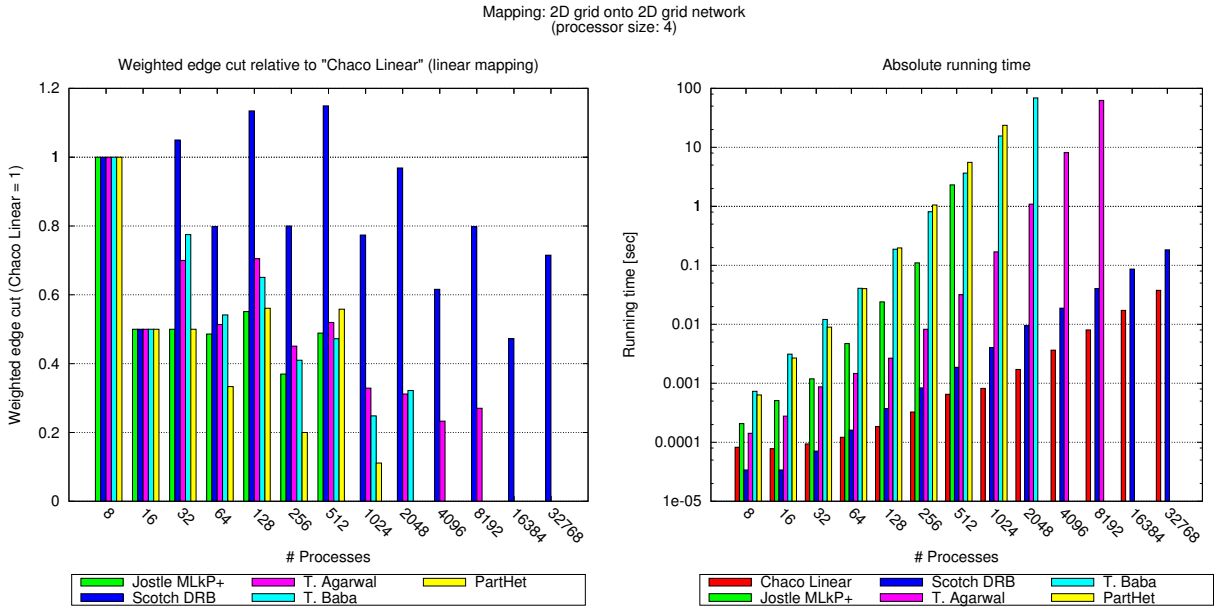


Figure A.16: Non-uniform communication cost mapping: 2D grid onto 2D grid network

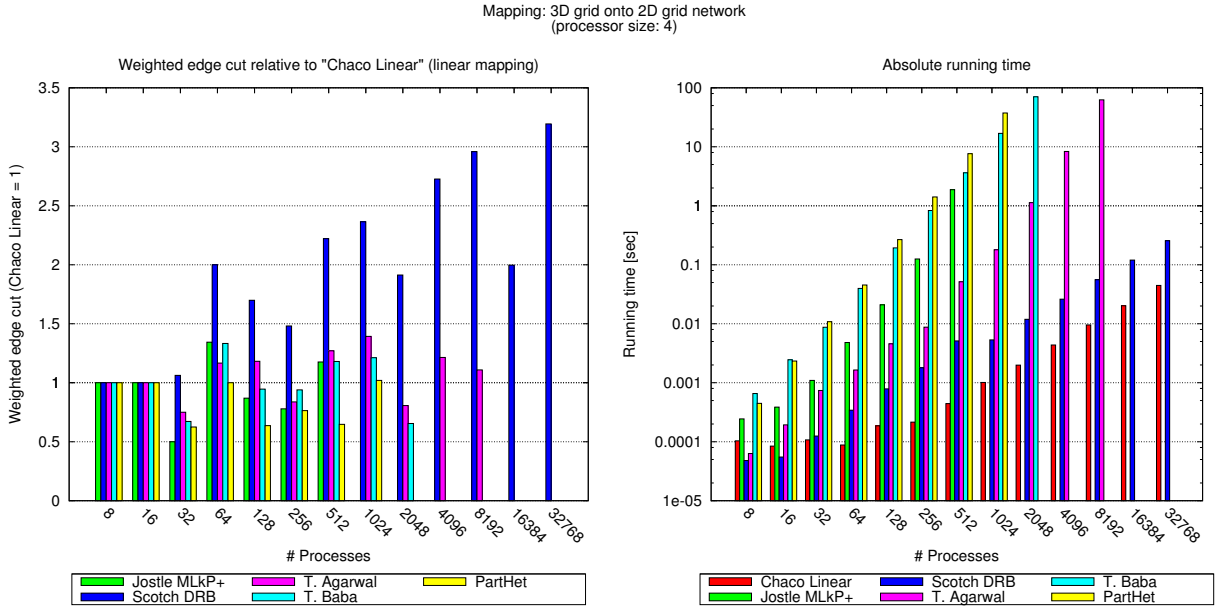


Figure A.17: Non-uniform communication cost mapping: 3D grid onto 2D grid network

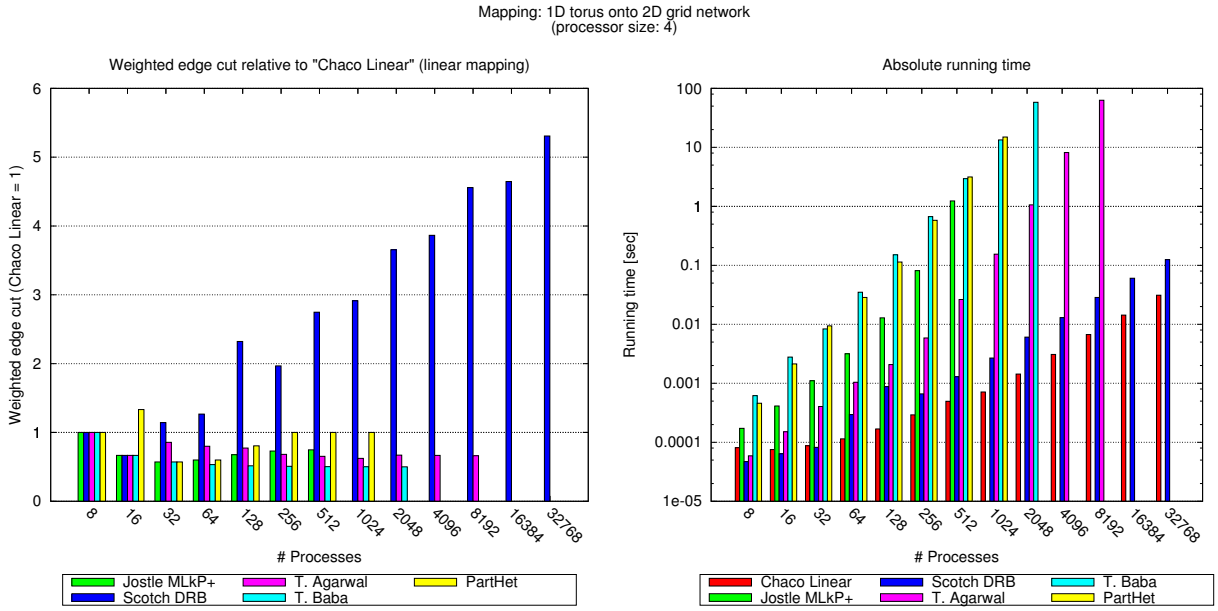


Figure A.18: Non-uniform communication cost mapping: 1D torus onto 2D grid network

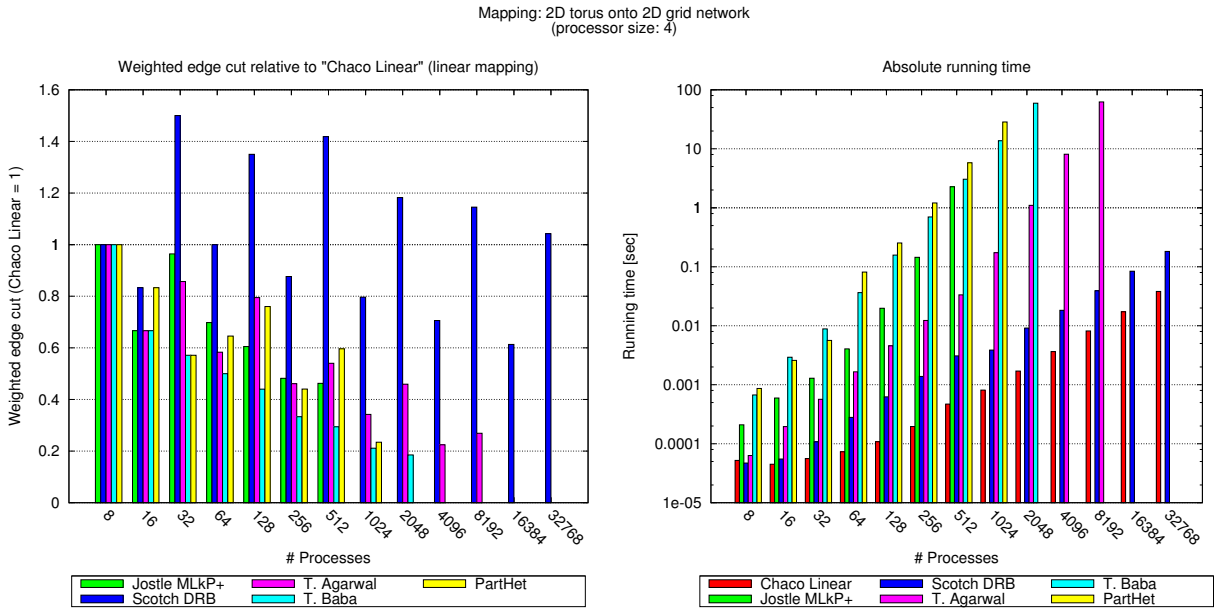


Figure A.19: Non-uniform communication cost mapping: 2D torus onto 2D grid network

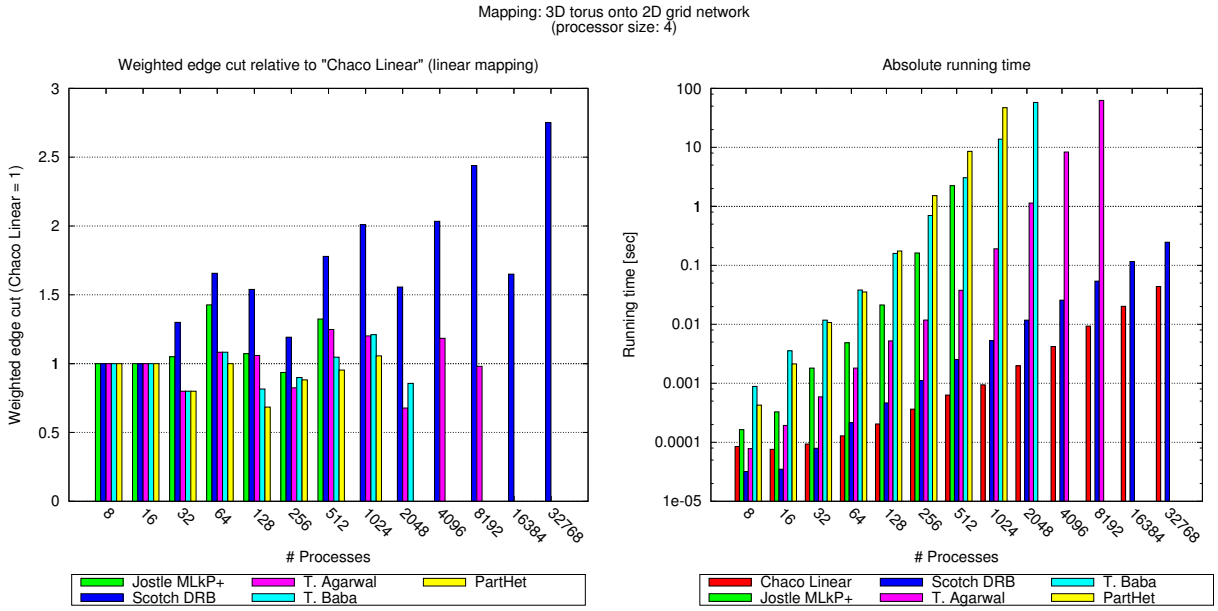


Figure A.20: Non-uniform communication cost mapping: 3D torus onto 2D grid network

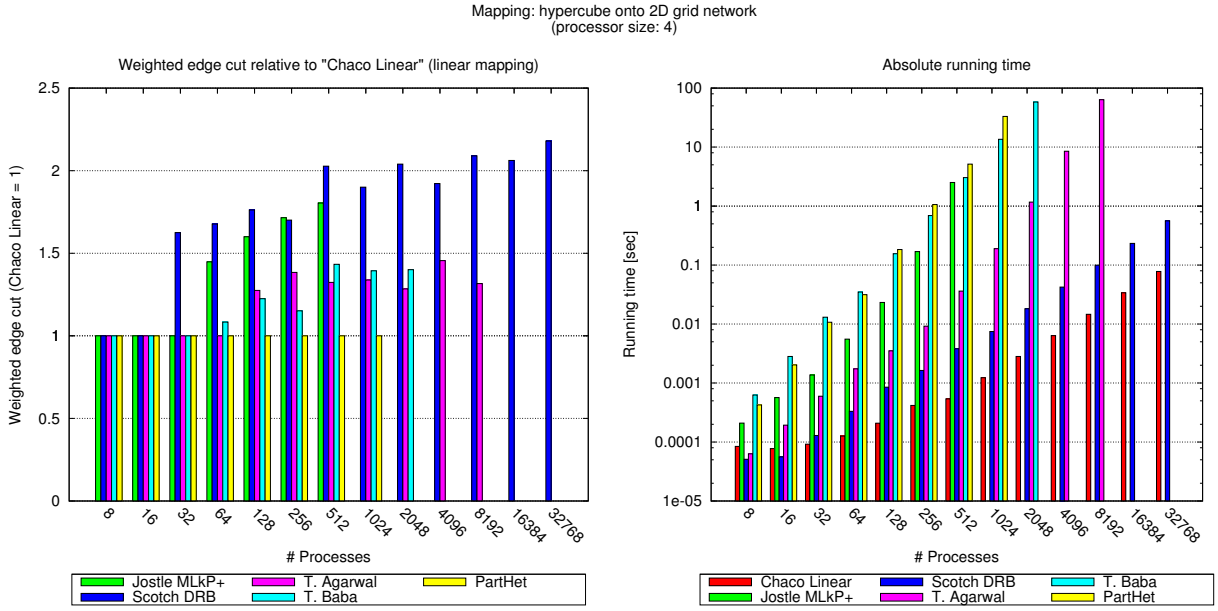


Figure A.21: Non-uniform communication cost mapping: hypercube onto 2D grid network

A.4 3D Grid Network Topology

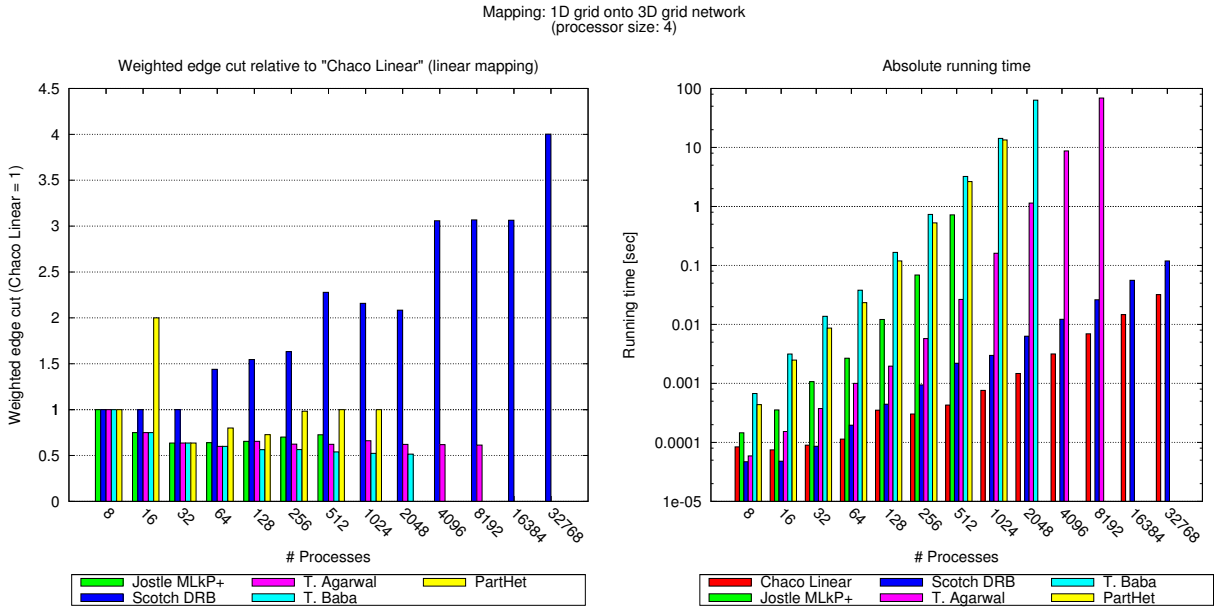


Figure A.22: Non-uniform communication cost mapping: 1D grid onto 3D grid network

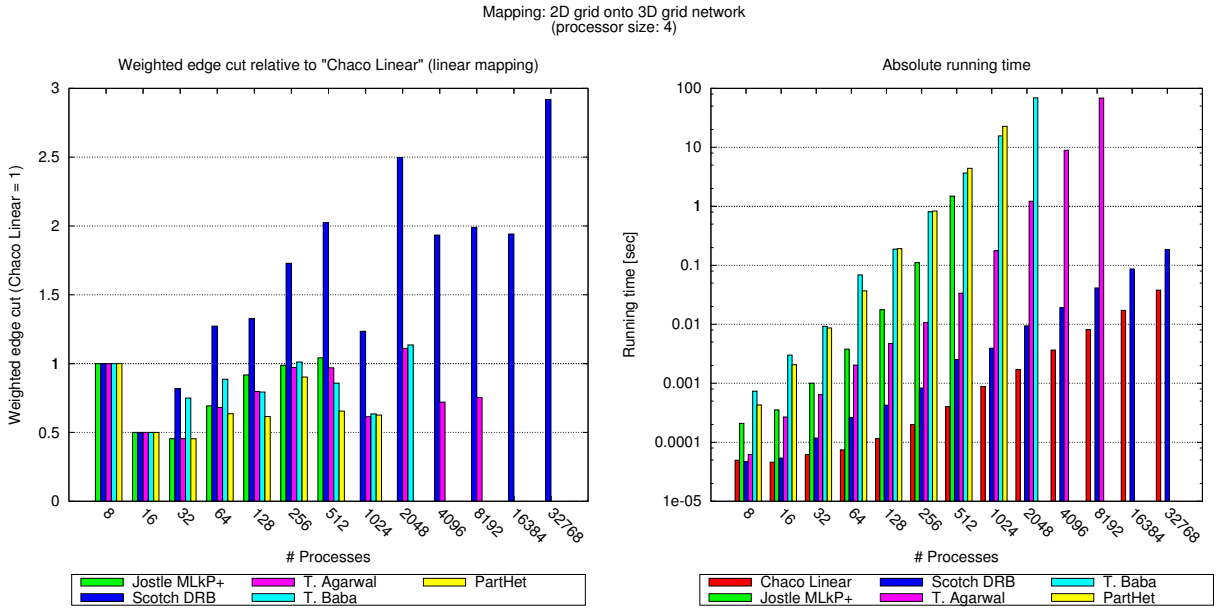


Figure A.23: Non-uniform communication cost mapping: 2D grid onto 3D grid network

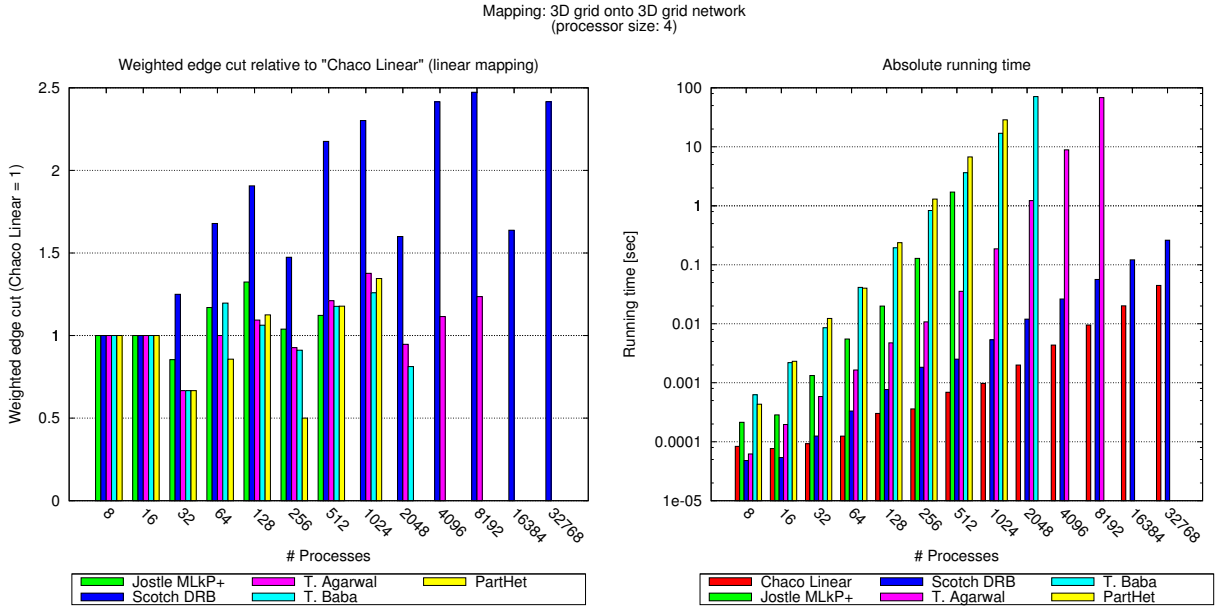


Figure A.24: Non-uniform communication cost mapping: 3D grid onto 3D grid network

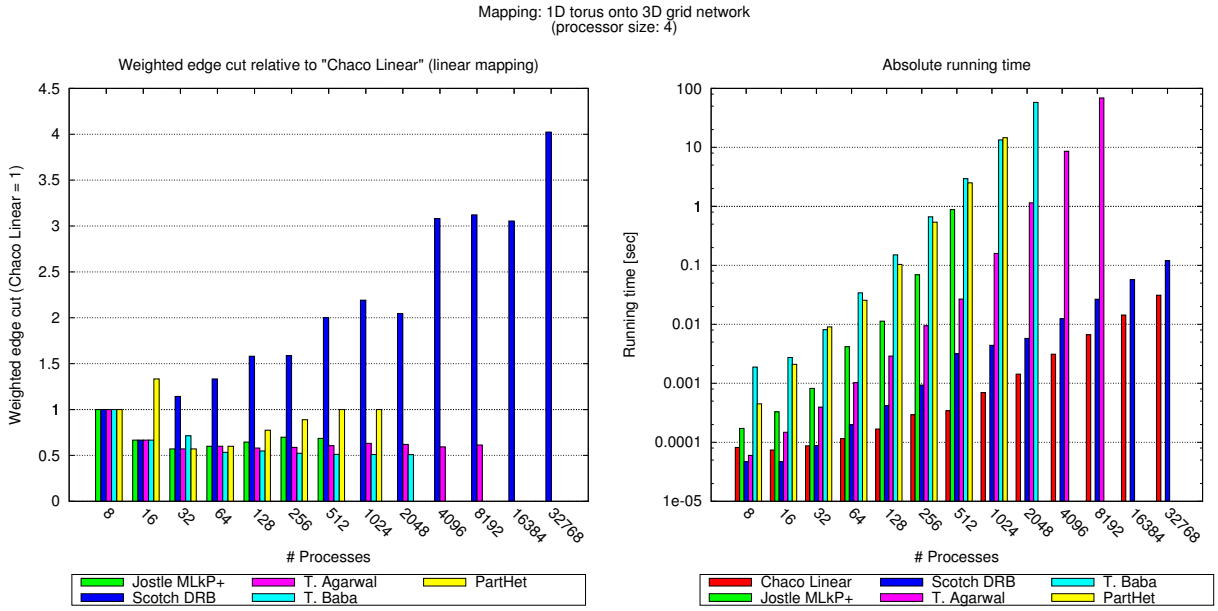


Figure A.25: Non-uniform communication cost mapping: 1D torus onto 3D grid network

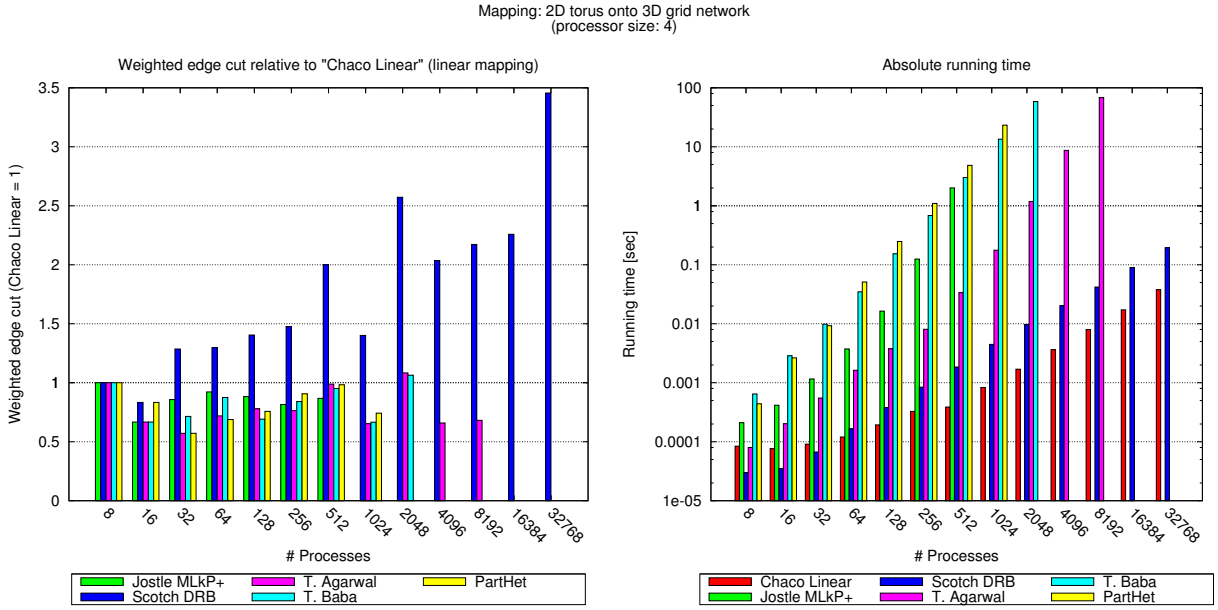


Figure A.26: Non-uniform communication cost mapping: 2D torus onto 3D grid network

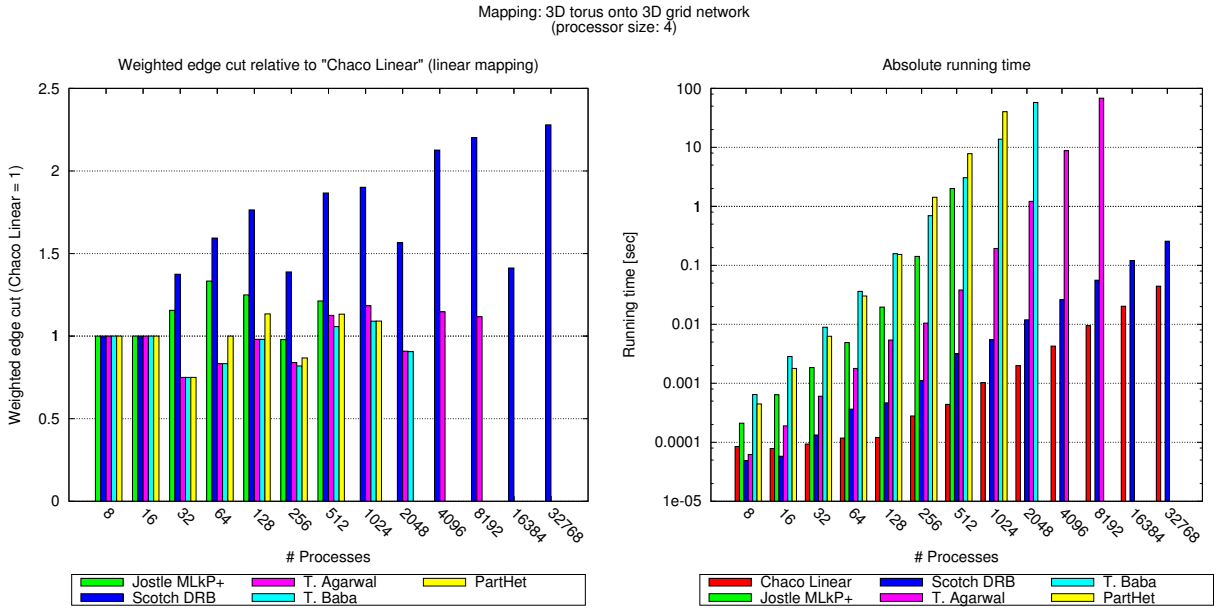


Figure A.27: Non-uniform communication cost mapping: 3D torus onto 3D grid network

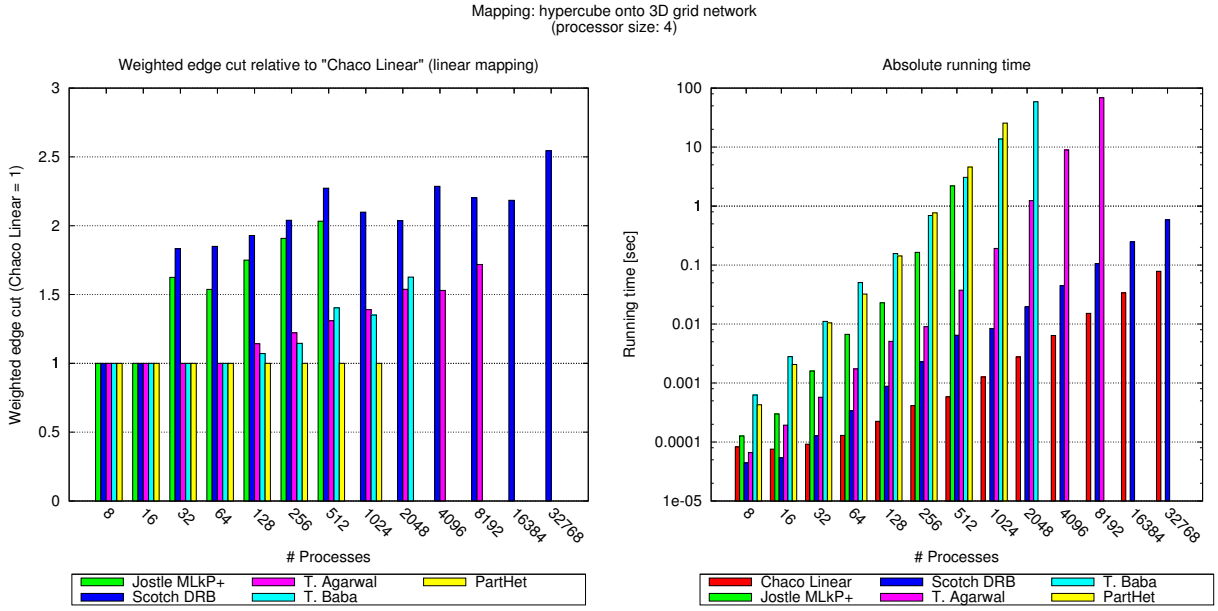


Figure A.28: Non-uniform communication cost mapping: hypercube onto 3D grid network

A.5 1D Torus Network Topology

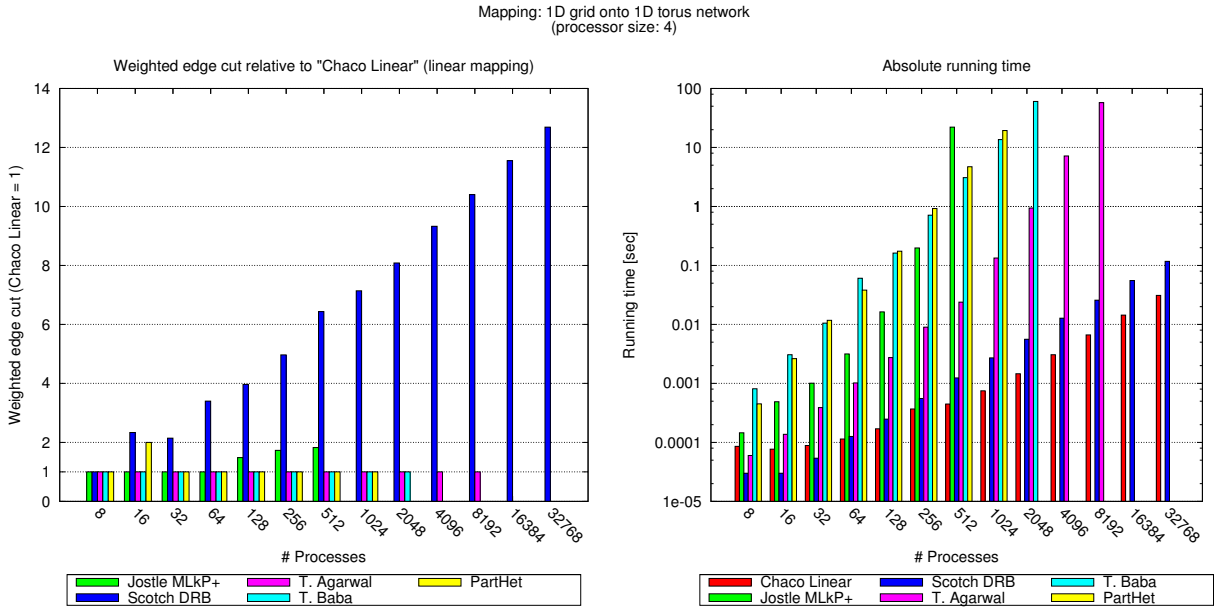


Figure A.29: Non-uniform communication cost mapping: 1D grid onto 1D torus network

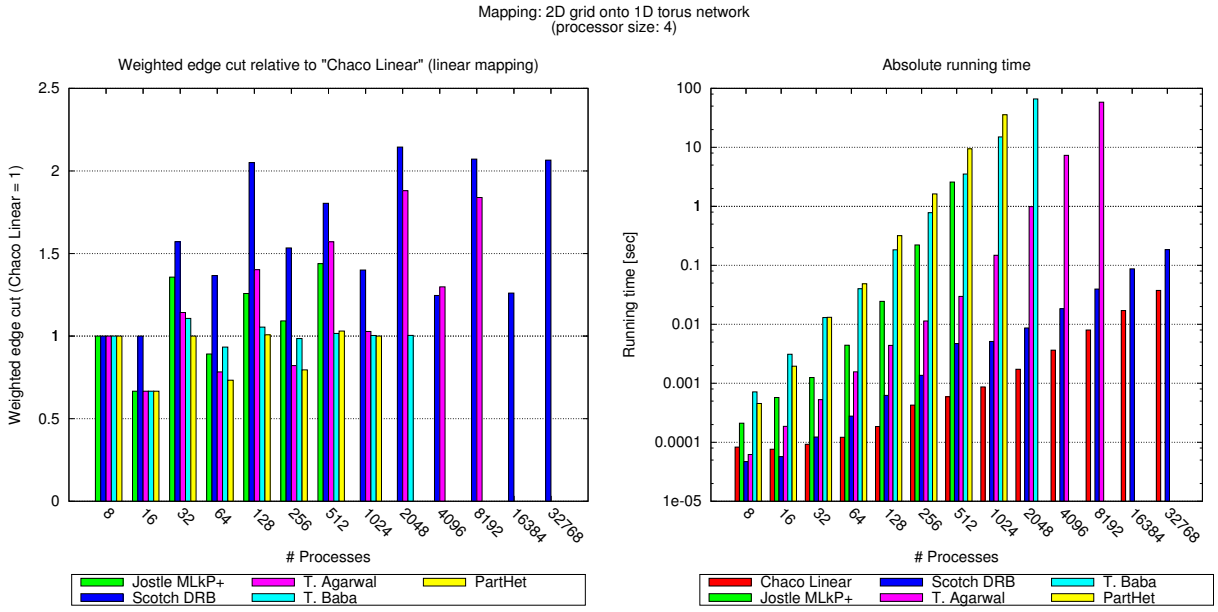


Figure A.30: Non-uniform communication cost mapping: 2D grid onto 1D torus network

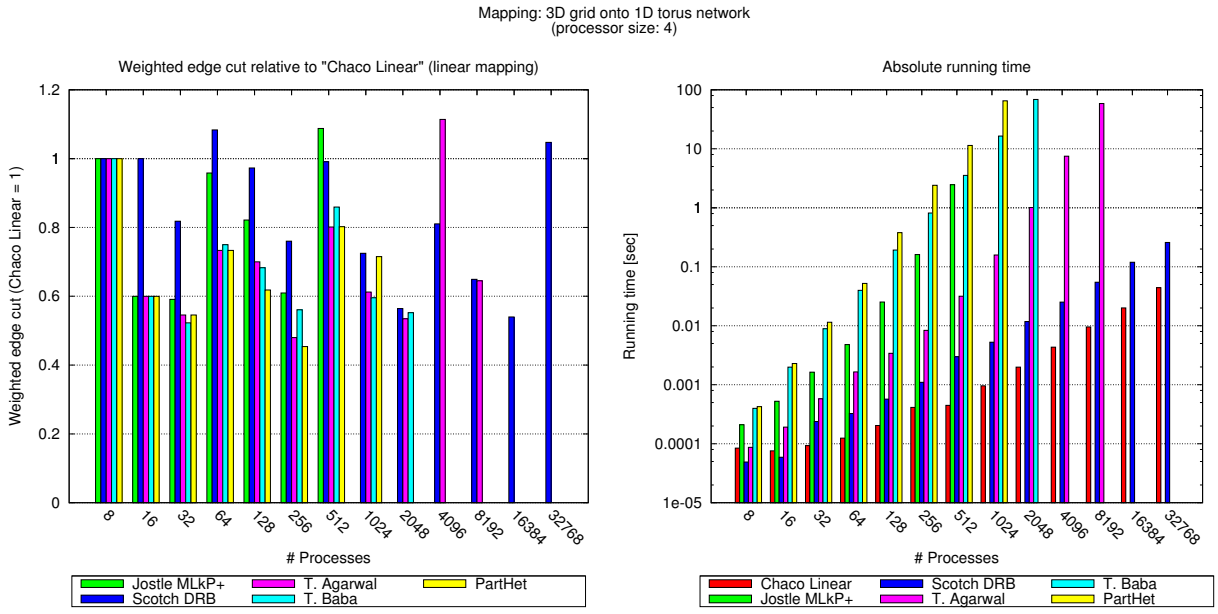


Figure A.31: Non-uniform communication cost mapping: 3D grid onto 1D torus network

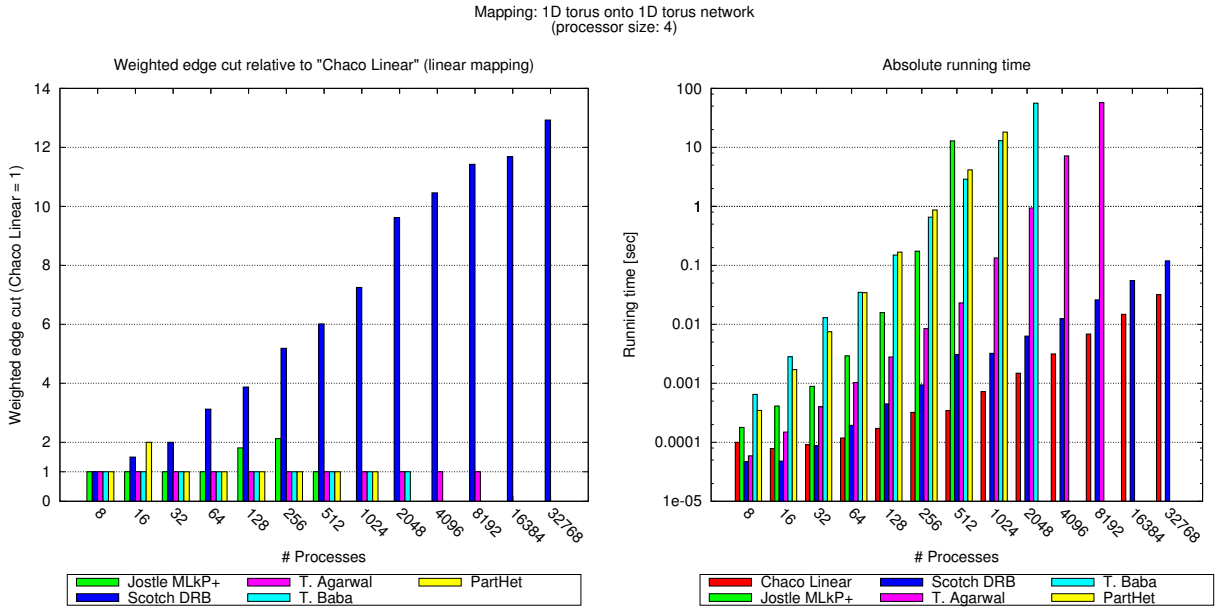


Figure A.32: Non-uniform communication cost mapping: 1D torus onto 1D torus network

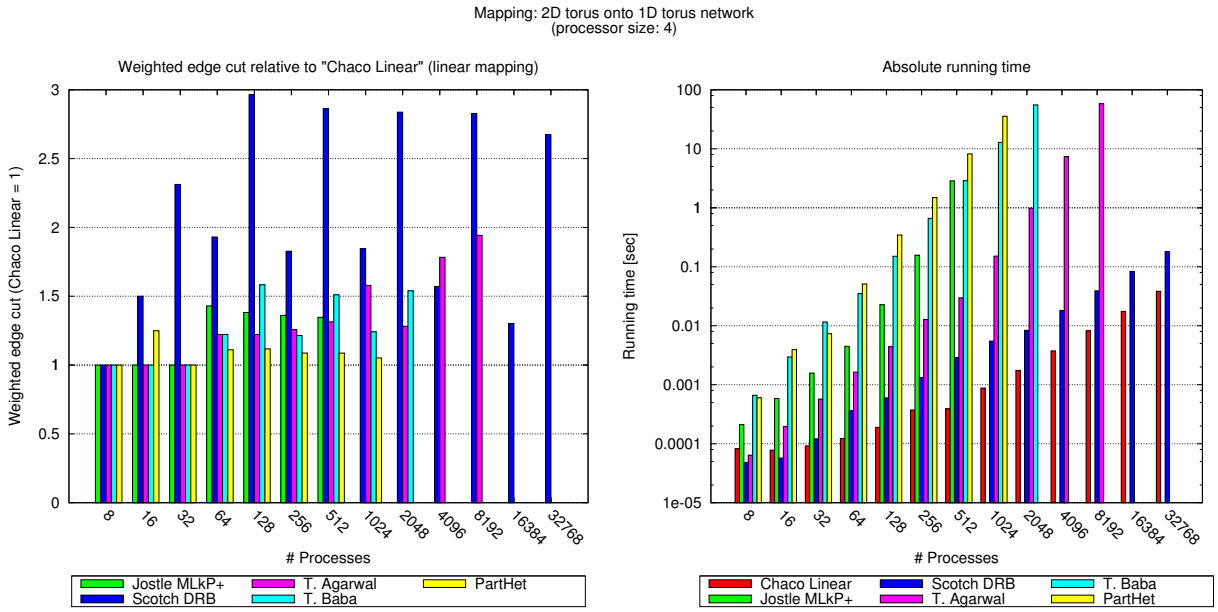


Figure A.33: Non-uniform communication cost mapping: 2D torus onto 1D torus network

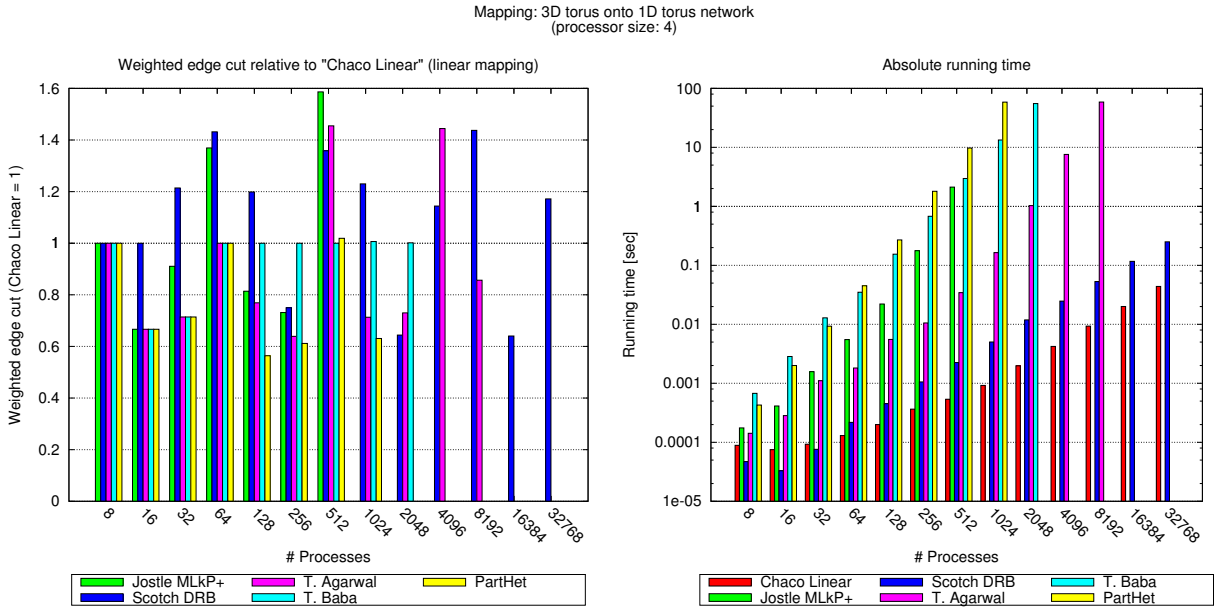


Figure A.34: Non-uniform communication cost mapping: 3D torus onto 1D torus network

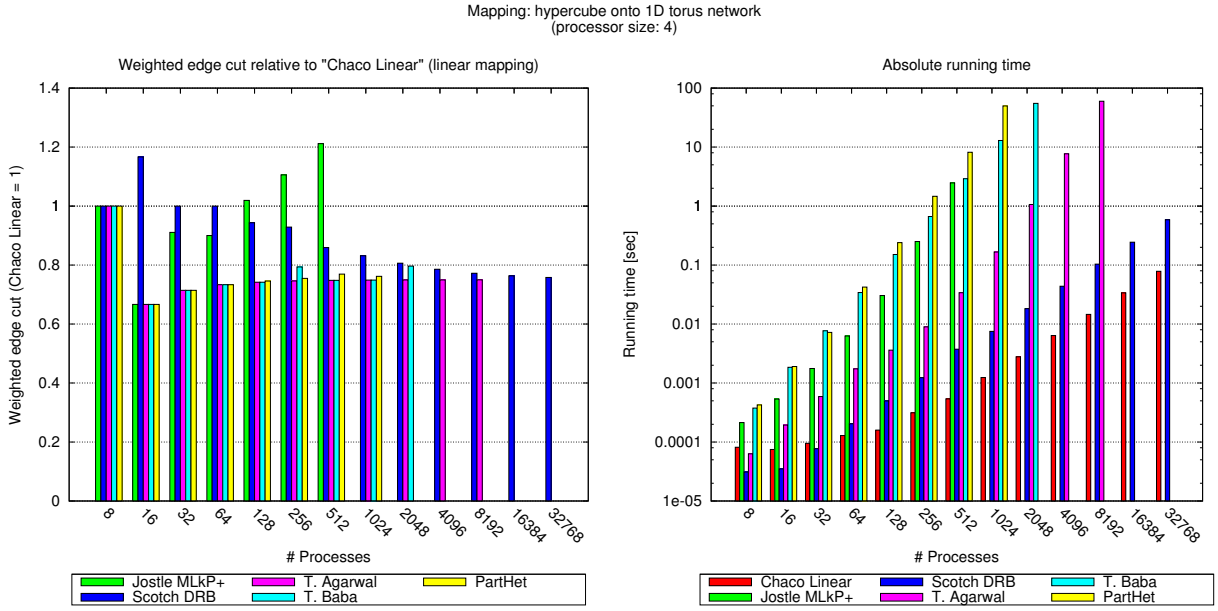


Figure A.35: Non-uniform communication cost mapping: hypercube onto 1D torus network

A.6 2D Torus Network Topology

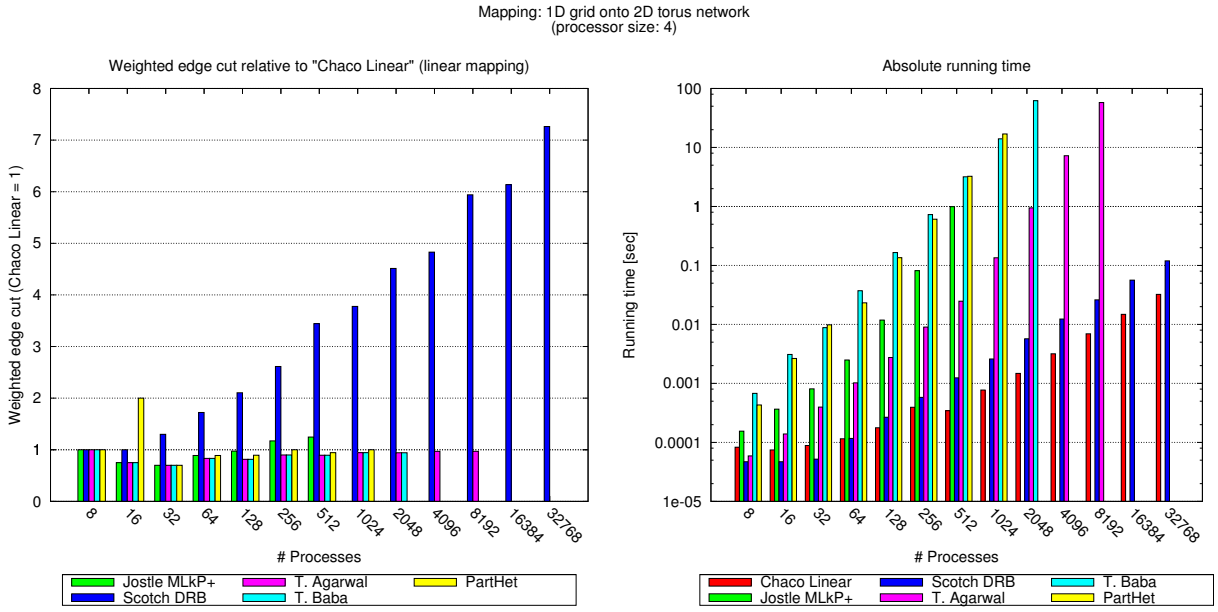


Figure A.36: Non-uniform communication cost mapping: 1D grid onto 2D torus network

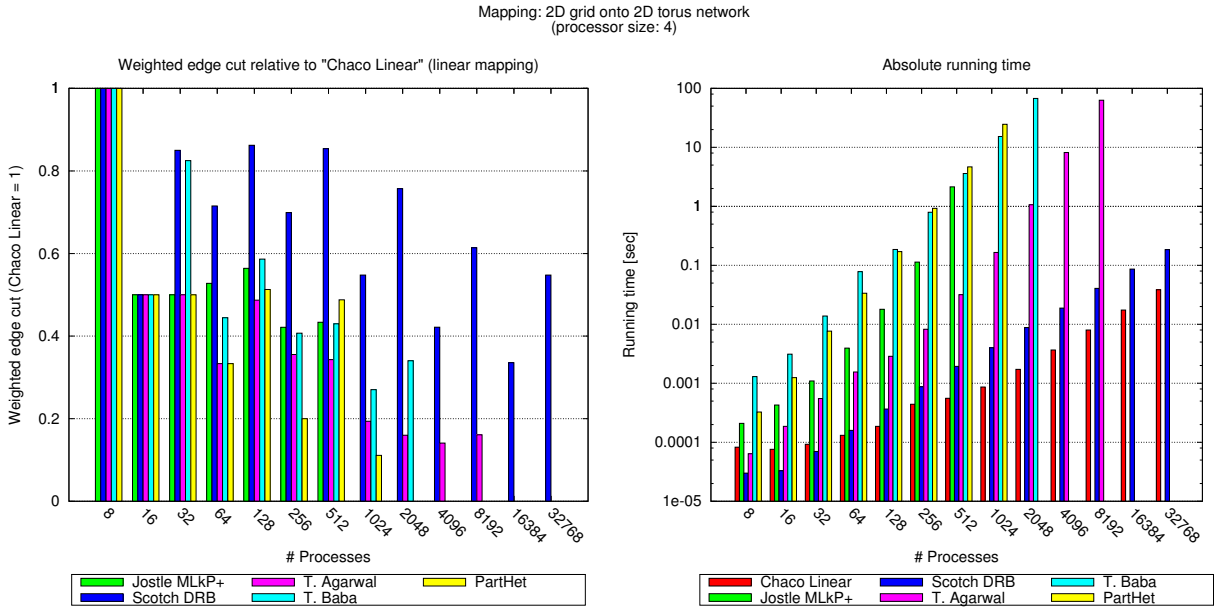


Figure A.37: Non-uniform communication cost mapping: 2D grid onto 2D torus network

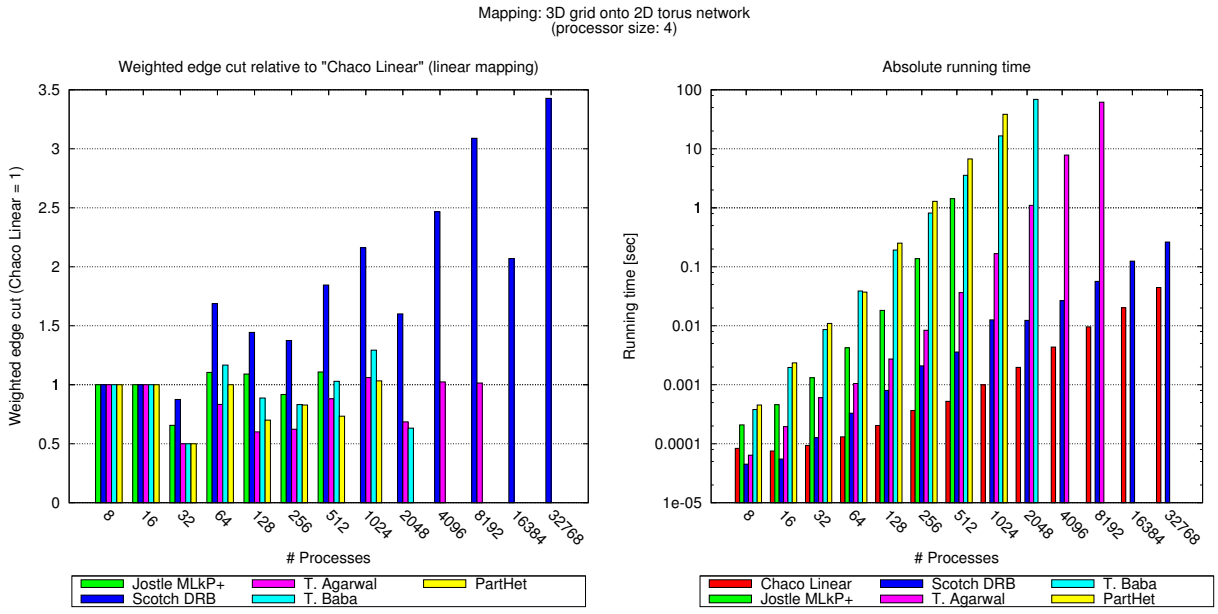


Figure A.38: Non-uniform communication cost mapping: 3D grid onto 2D torus network

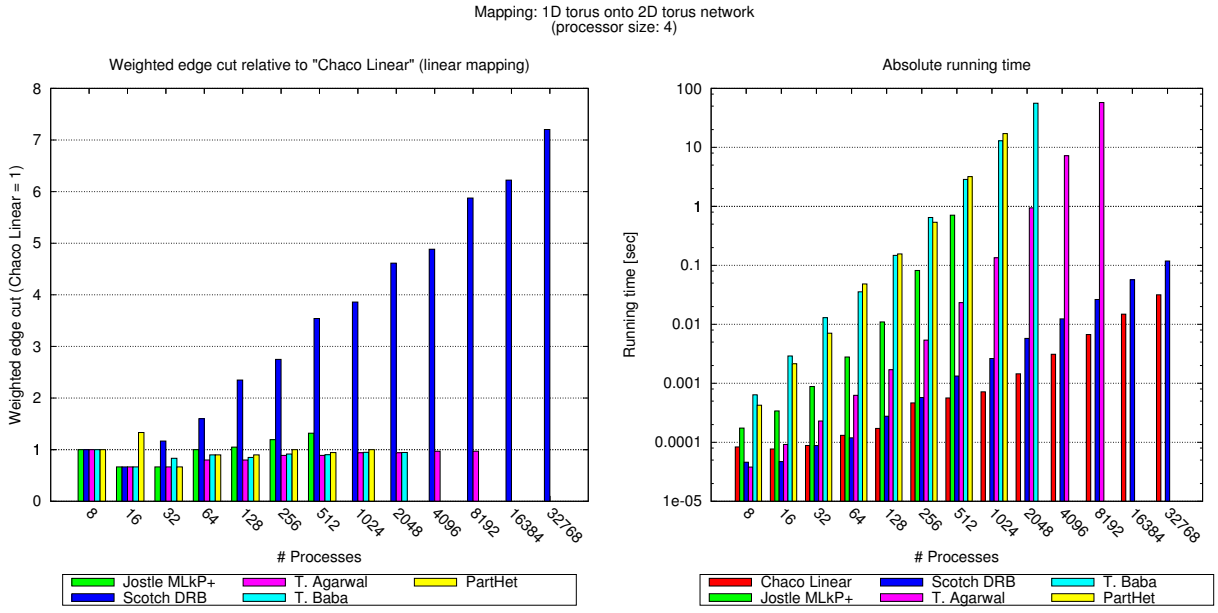


Figure A.39: Non-uniform communication cost mapping: 1D torus onto 2D torus network

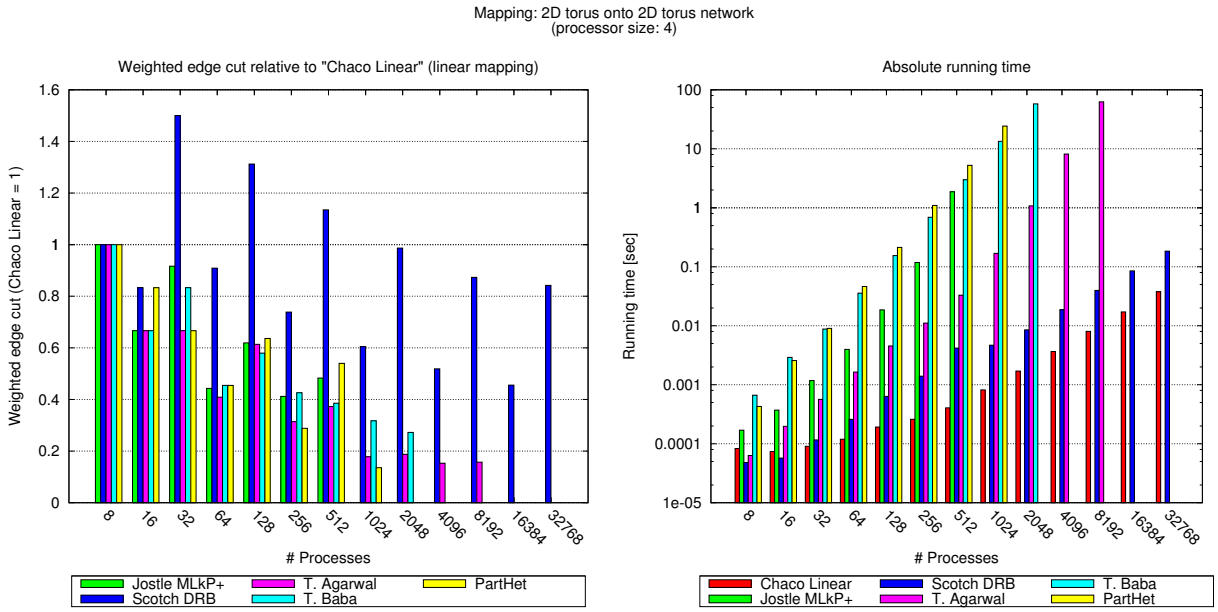


Figure A.40: Non-uniform communication cost mapping: 2D torus onto 2D torus network

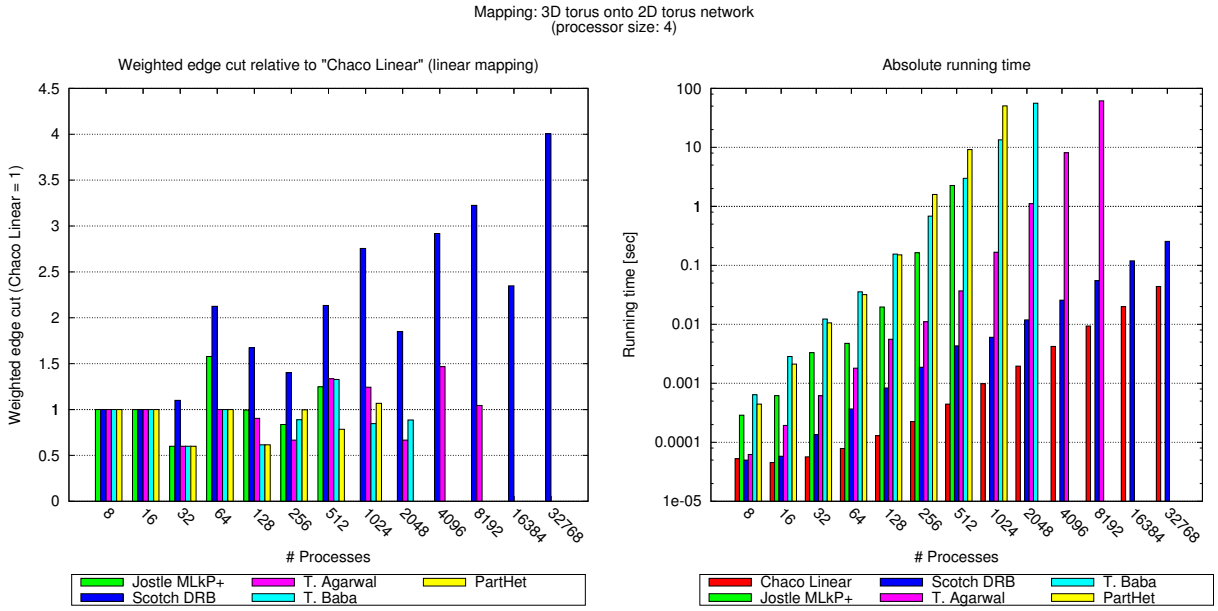


Figure A.41: Non-uniform communication cost mapping: 3D torus onto 2D torus network

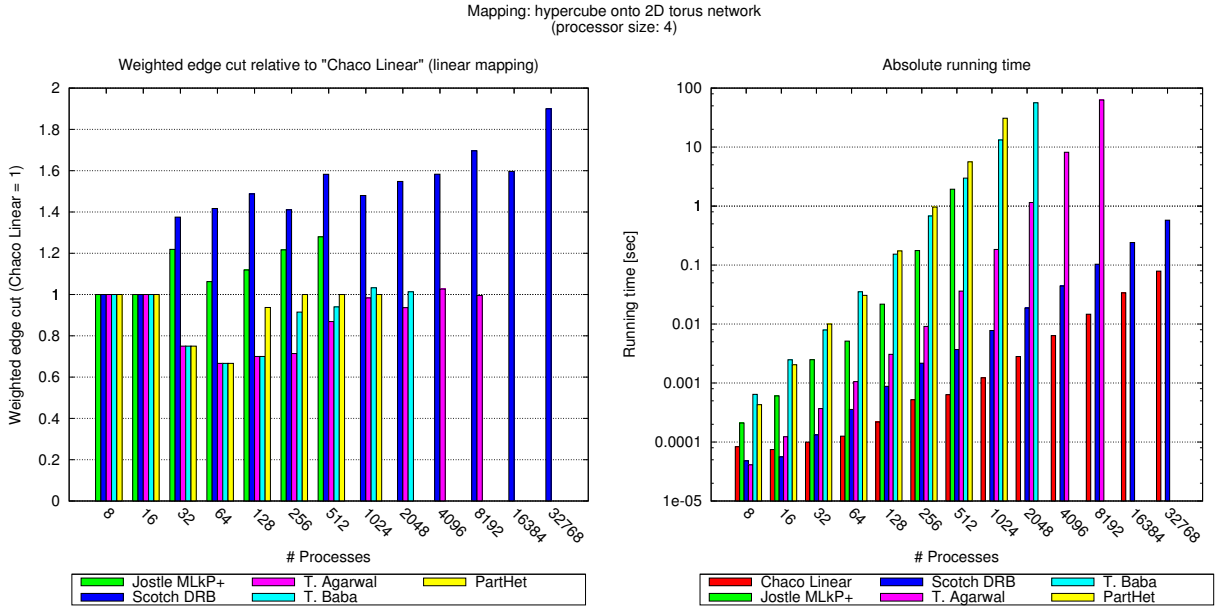


Figure A.42: Non-uniform communication cost mapping: hypercube onto 2D torus network

A.7 3D Torus Network Topology

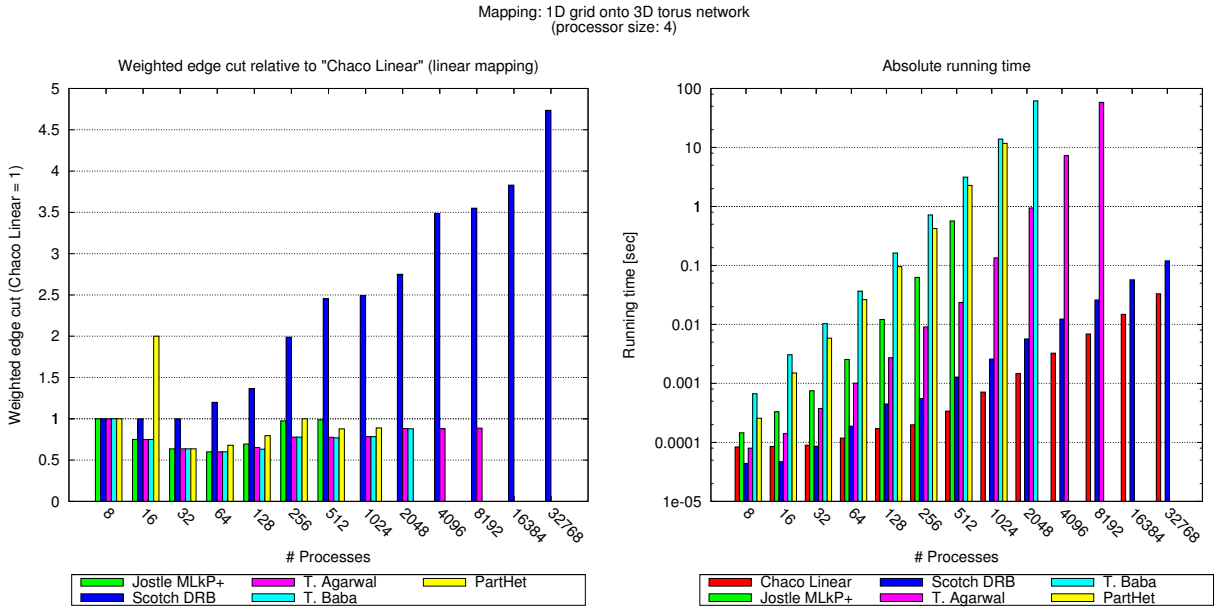


Figure A.43: Non-uniform communication cost mapping: 1D grid onto 3D torus network

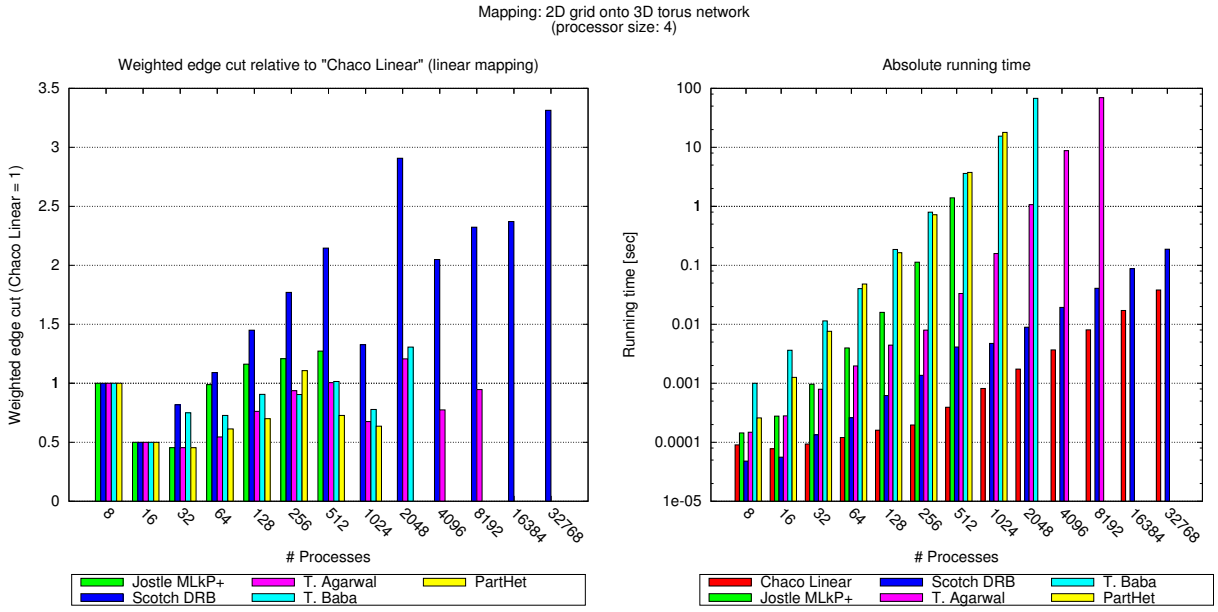


Figure A.44: Non-uniform communication cost mapping: 2D grid onto 3D torus network

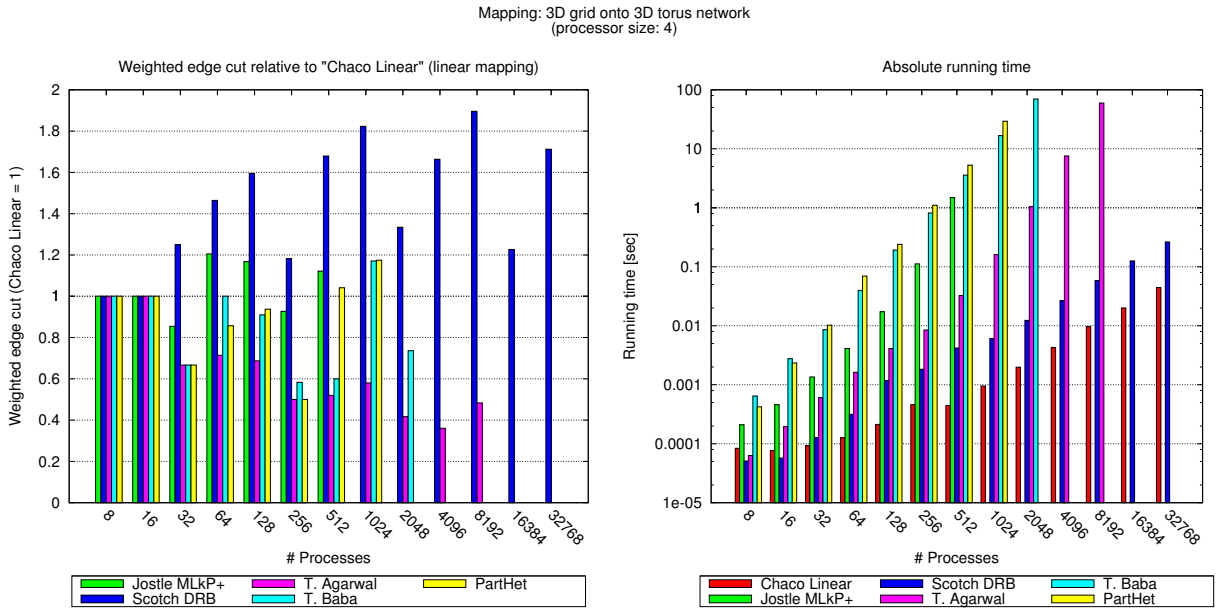


Figure A.45: Non-uniform communication cost mapping: 3D grid onto 3D torus network

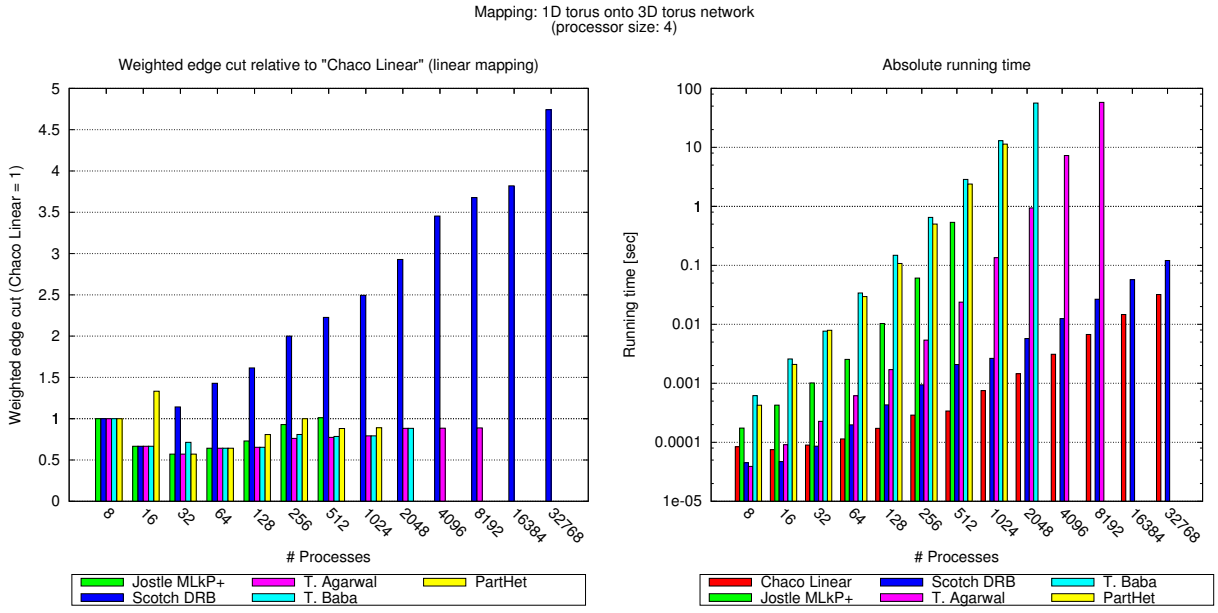


Figure A.46: Non-uniform communication cost mapping: 1D torus onto 3D torus network

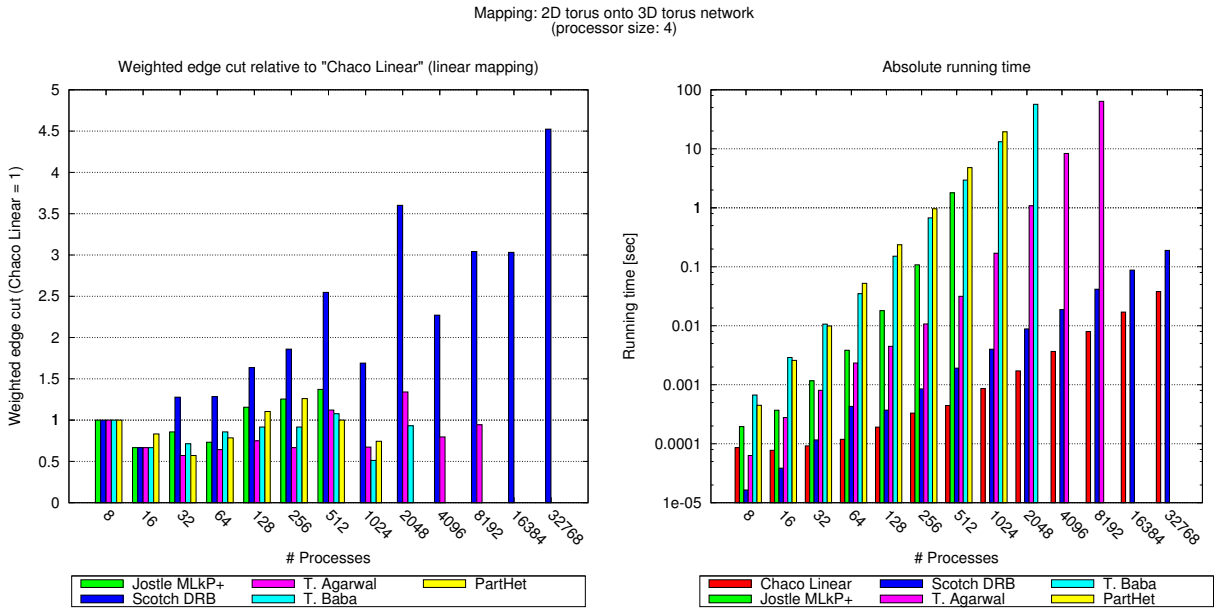


Figure A.47: Non-uniform communication cost mapping: 2D torus onto 3D torus network

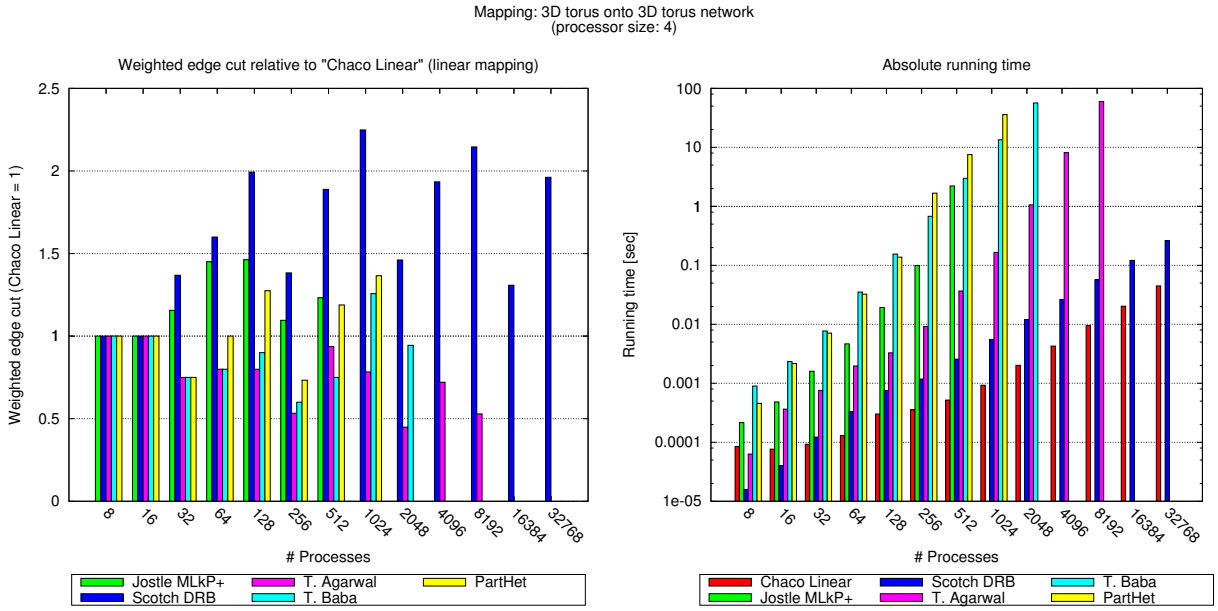


Figure A.48: Non-uniform communication cost mapping: 3D torus onto 3D torus network

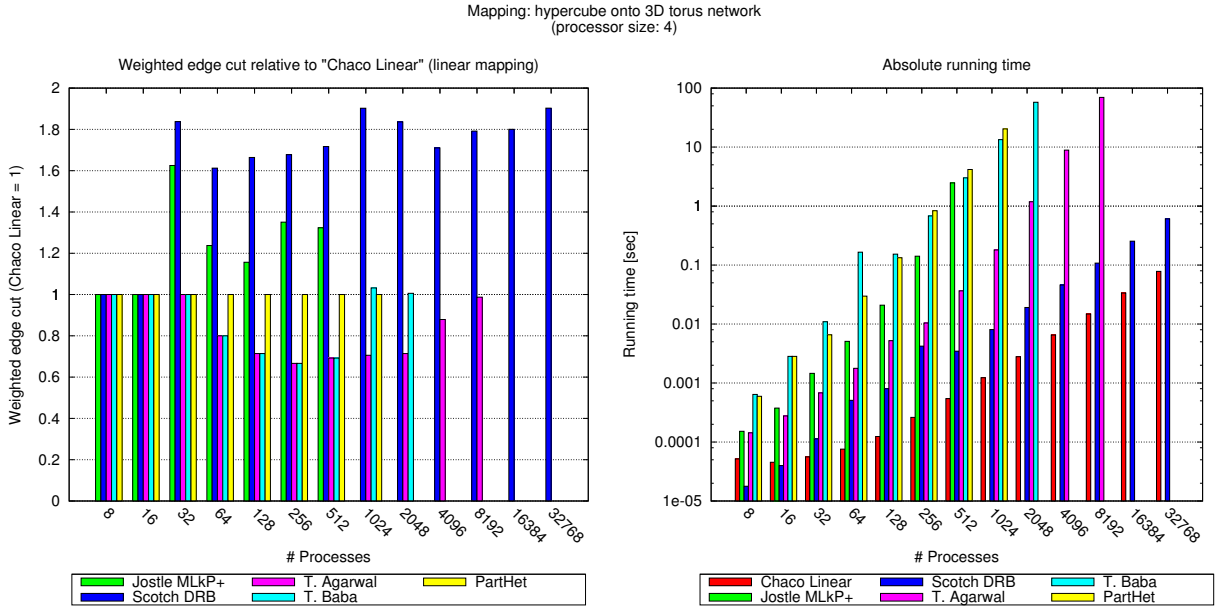


Figure A.49: Non-uniform communication cost mapping: hypercube onto 3D torus network

A.8 Hypercube Network Topology

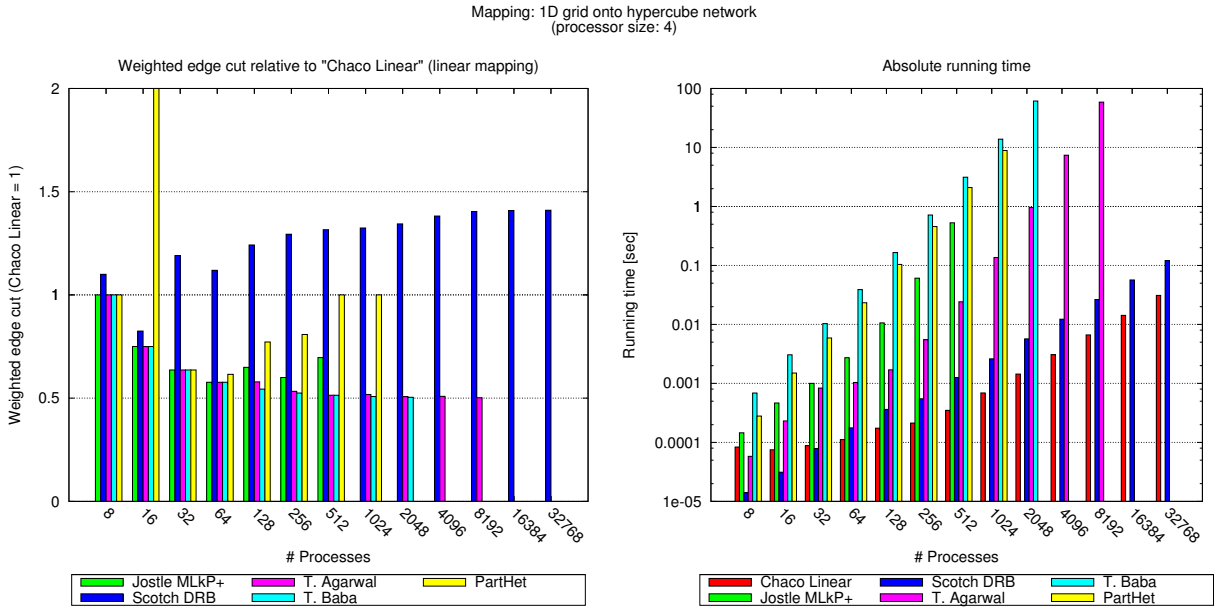


Figure A.50: Non-uniform communication cost mapping: 1D grid onto hypercube network

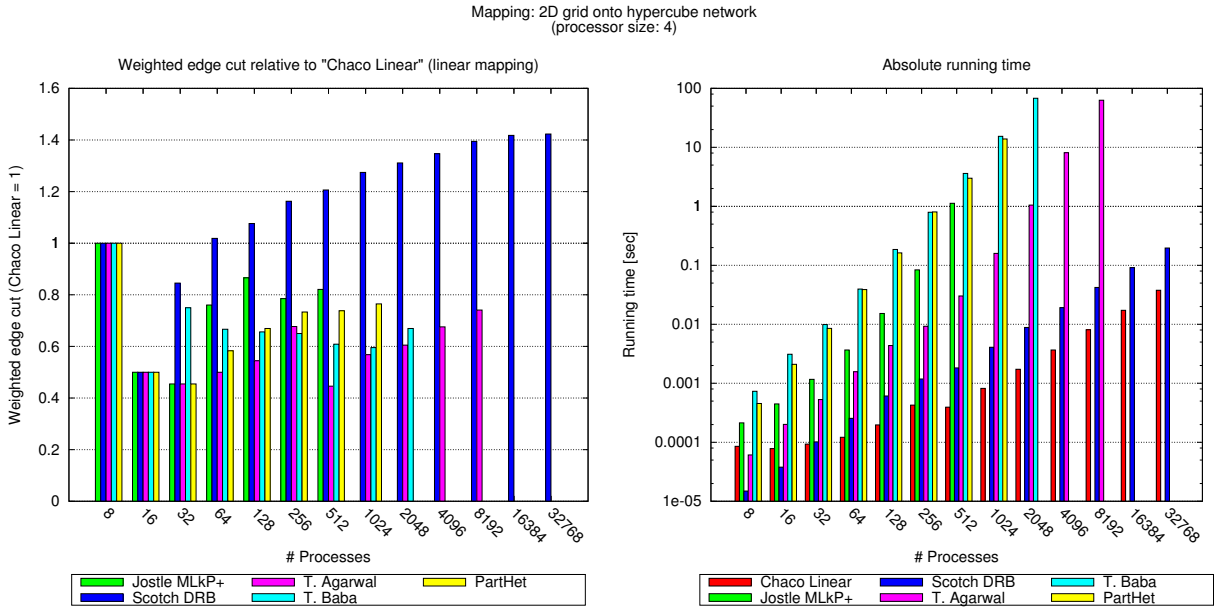


Figure A.51: Non-uniform communication cost mapping: 2D grid onto hypercube network

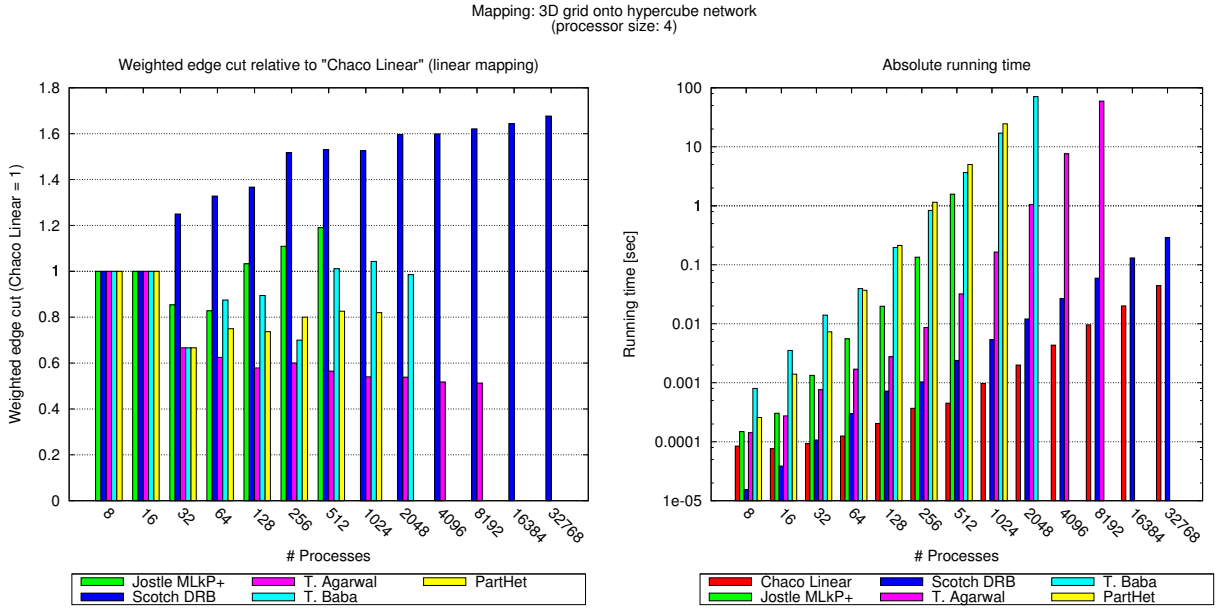


Figure A.52: Non-uniform communication cost mapping: 3D grid onto hypercube network

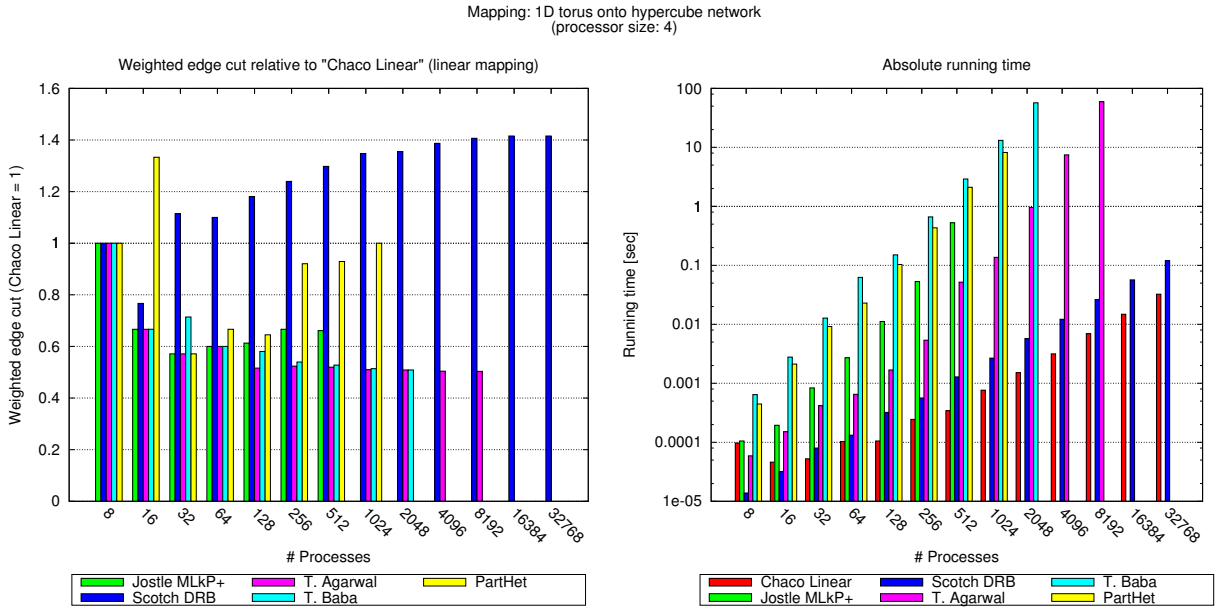


Figure A.53: Non-uniform communication cost mapping: 1D torus onto hypercube network

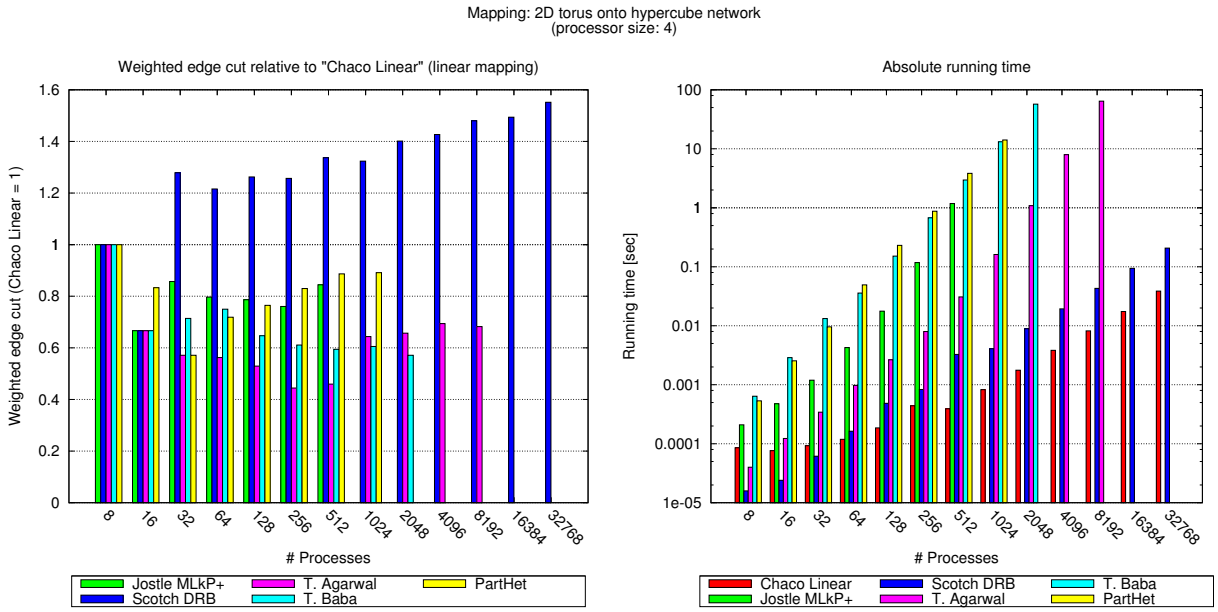


Figure A.54: Non-uniform communication cost mapping: 2D torus onto hypercube network

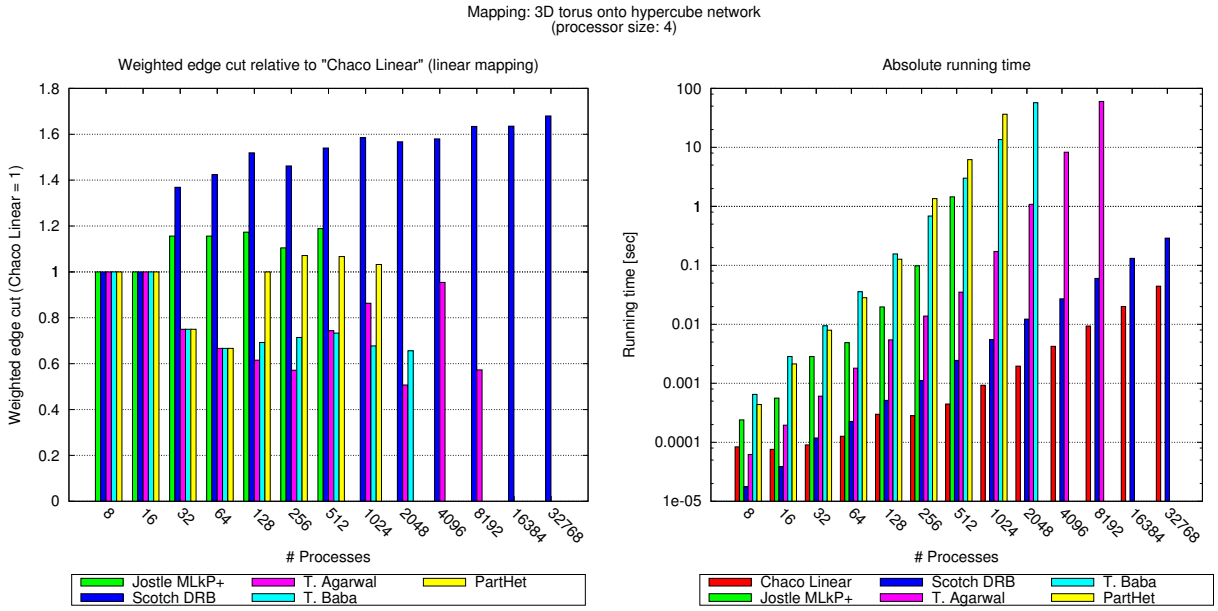


Figure A.55: Non-uniform communication cost mapping: 3D torus onto hypercube network

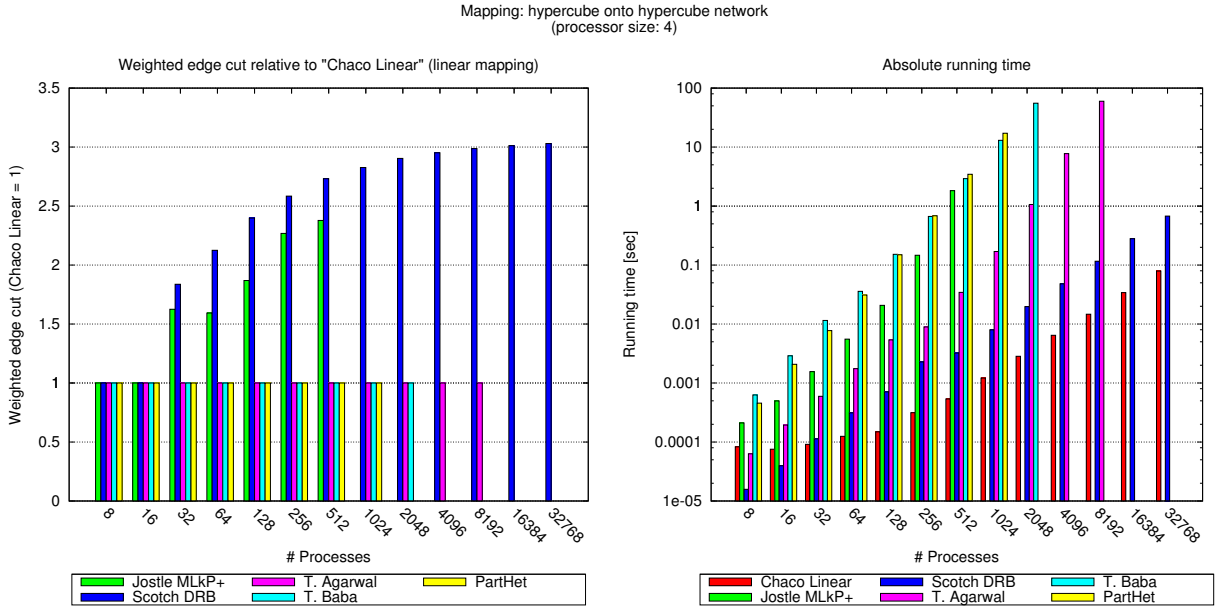


Figure A.56: Non-uniform communication cost mapping: hypercube onto hypercube network

A.9 Binary Tree Network Topology

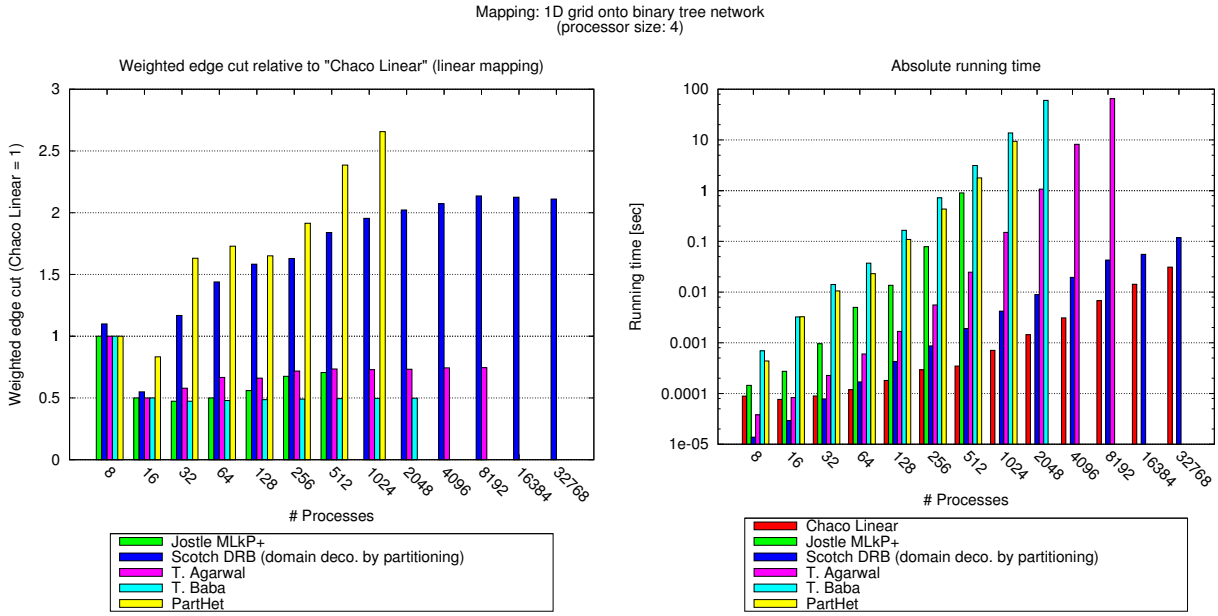


Figure A.57: Non-uniform communication cost mapping: 1D grid onto binary tree network

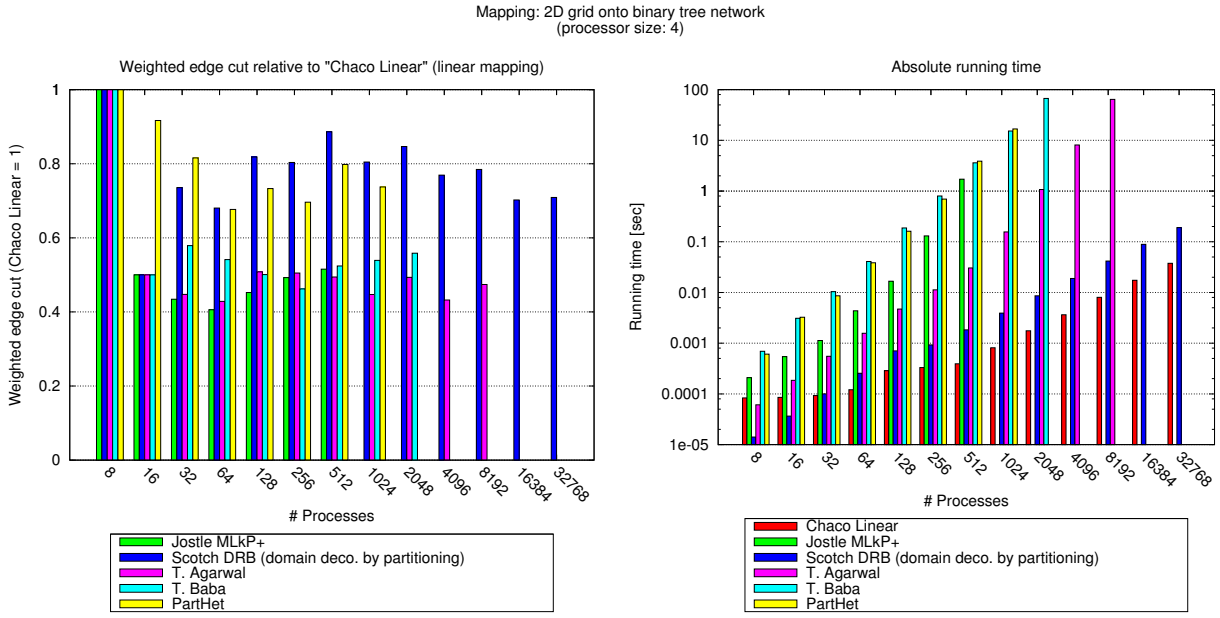


Figure A.58: Non-uniform communication cost mapping: 2D grid onto binary tree network

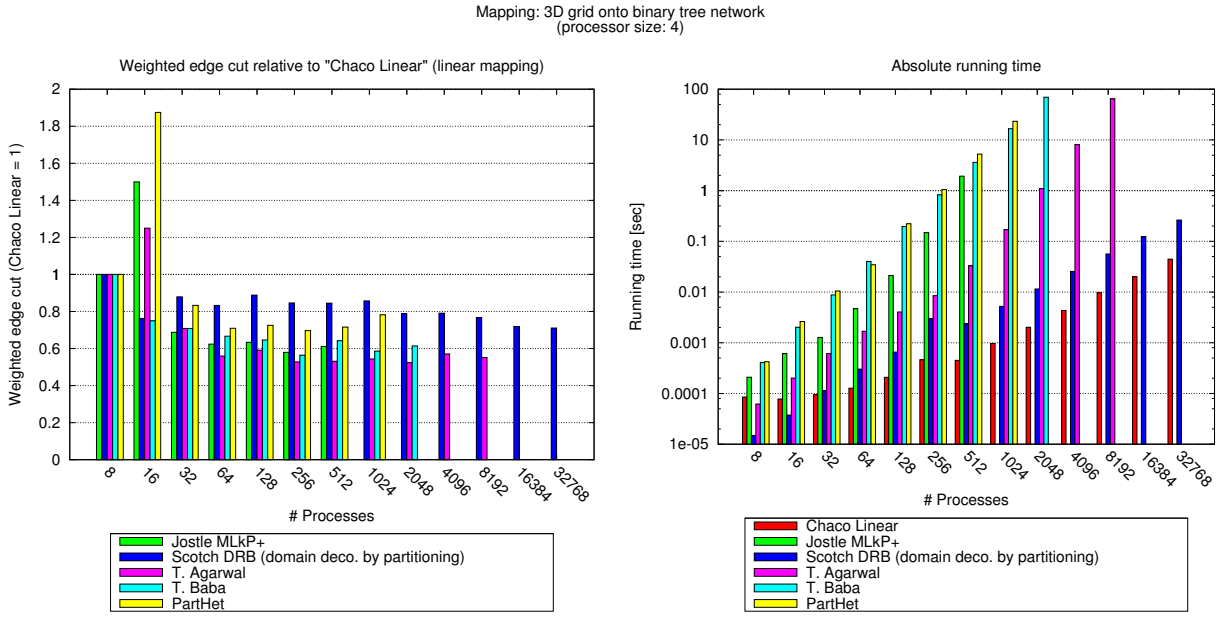


Figure A.59: Non-uniform communication cost mapping: 3D grid onto binary tree network

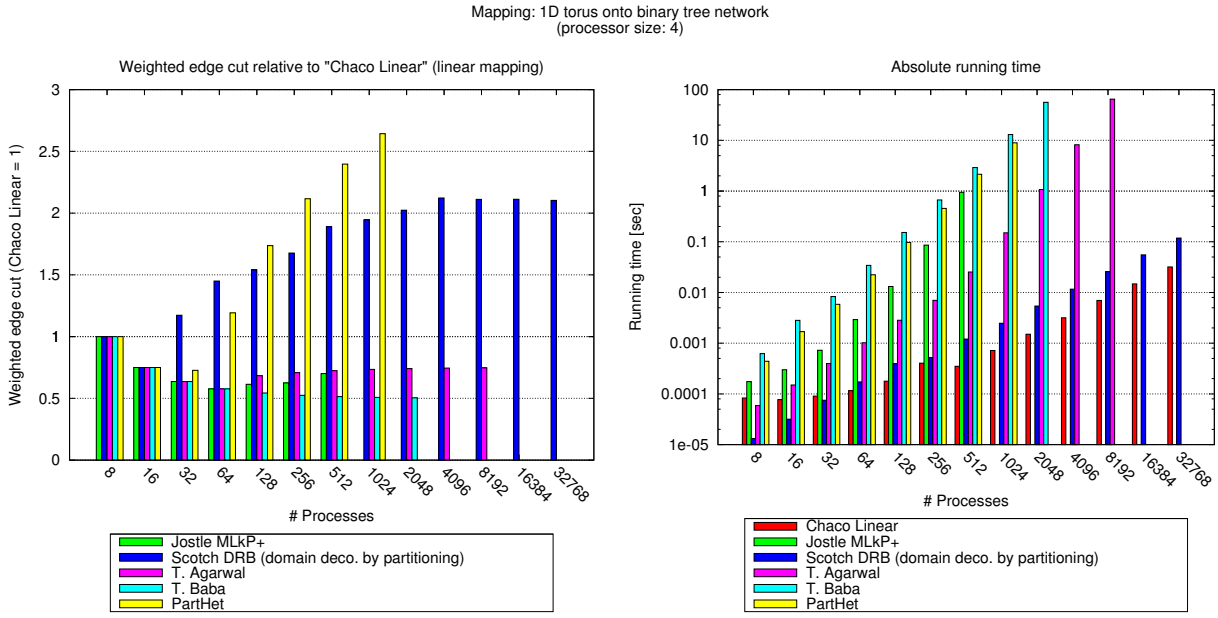


Figure A.60: Non-uniform communication cost mapping: 1D torus onto binary tree network

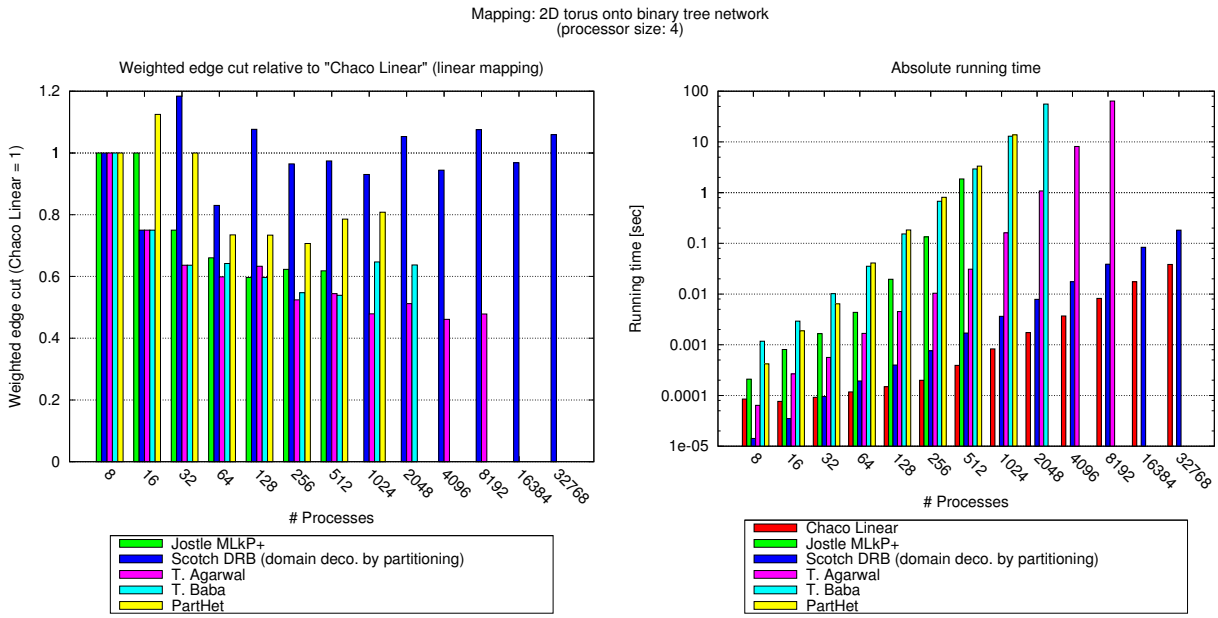


Figure A.61: Non-uniform communication cost mapping: 2D torus onto binary tree network

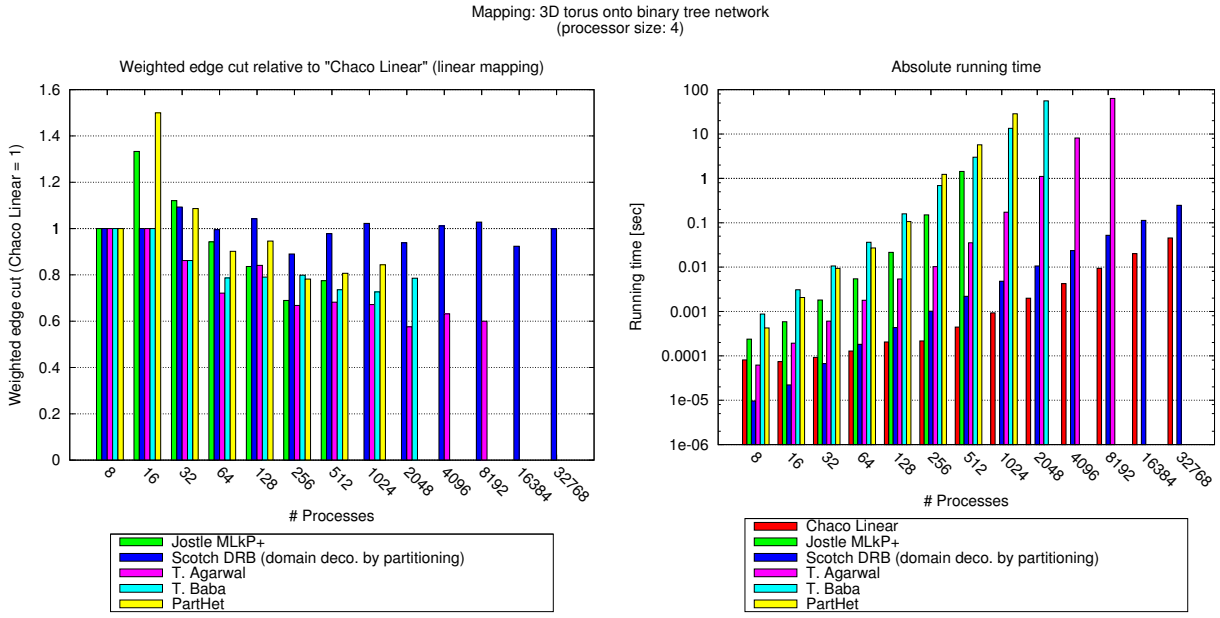


Figure A.62: Non-uniform communication cost mapping: 3D torus onto binary tree network

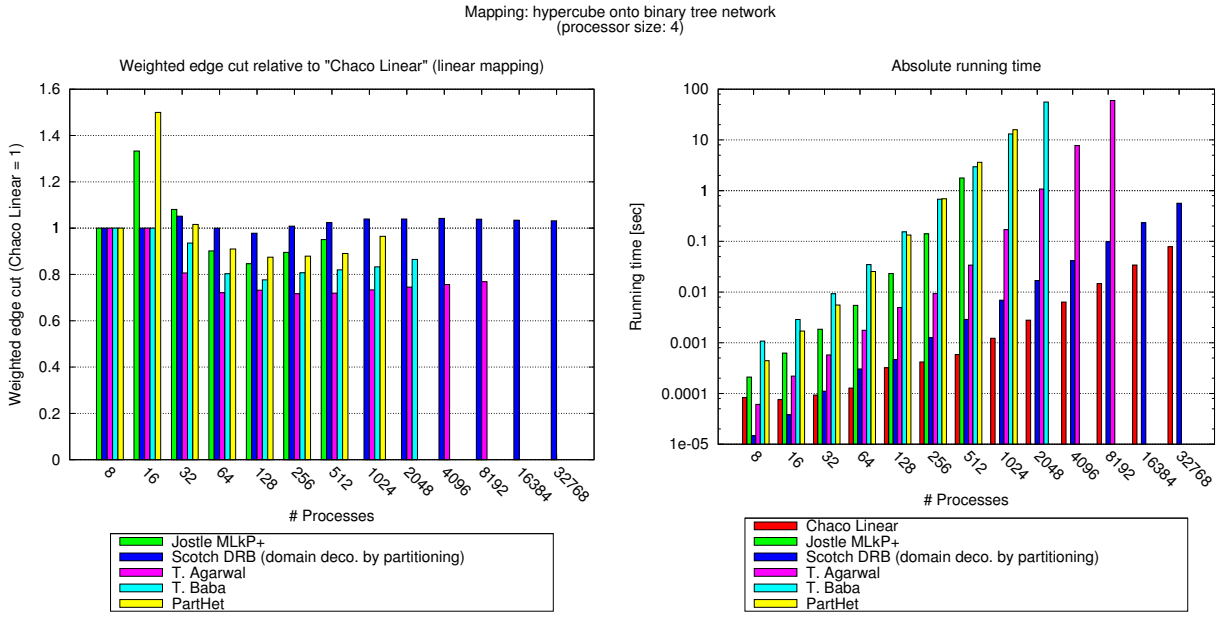


Figure A.63: Non-uniform communication cost mapping: hypercube onto binary tree network

Bibliography

[AAC+04] George Almási, Charles Archer, José G. Castaños, C. Christopher Erway, Philip Heidelberger, Xavier Martorell, José E. Moreira, Kurt Pinnow, Joe Ratterman, Nils Smeds, Burkhard D. Steinmacher-Burrow, William Gropp, and Brian R. Toonen. Implementing MPI on the BlueGene/L supercomputer. In *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 833–845. Springer, 2004.

[AISS97] Alexandrov, Ionescu, Schauser, and Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *JPDC: Journal of Parallel and Distributed Computing*, 44, 1997.

[ASK06] T. Agarwal, A. Sharma, and L. V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)*, Rhodes Island, Greece, April 2006. IEEE Computer Society.

[BGH+05] Gyan Bhanot, Alan Gara, Philip Heidelberger, Eoin Lawless, James C. Sexton, and Robert Walkup. Optimizing task layout on the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2-3):489–500, 2005.

[BIY90] Takanobu Baba, Yoshifumi Iwamoto, and Tsutomu Yoshinaga. A network-topology independent task allocation strategy for parallel computers. In *SC*, pages 878–887, 1990.

[BP04] J.L. Bosque and L.P. Perez. HLogGP: a new parallel computational model for heterogeneous clusters. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:403–410, 2004.

[BR84] R. E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17:169–174, 1984.

[CKP+93] Culler, Karp, Patterson, Sahay, Schauser, Santos, Subramonian, and von Eicken. LogP: Towards a realistic model of parallel computation. In *PPOPP: 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming: PPOPP*, 1993.

[CRC+02] Cortés, Ripoll, Cedó, Senar, and Luque. An asynchronous and iterative load balancing algorithm for discrete load model. *JPDC: Journal of Parallel and Distributed Computing*, 62, 2002.

[FBR87] Finke, Burkard, and Rendl. Quadratic assignment problems. *ANNALSDM: Annals of Discrete Mathematics*, 31, 1987.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. *19th Design Automation Conference*, pages 175–181, 1982.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference. Volume 2, The MPI-2 Extensions*. Scientific and Engineering Computation. MIT Press, second edition, 1998.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[GRTZ03] M. Golebiewski, H. Ritzdorf, J. L. Träff, and F. Zimmermann. The MPI/SX Implementation of MPI for NEC's SX-6 and Other NEC Platforms. *NEC Res. & Develop.*, 44(1), January 2003.

[Hat98] Takao Hatazaki. Rank Reordering Strategy for MPI Topology Creation Functions. In *PVM/MPI*, volume 1497 of *Lecture Notes in Computer Science*, pages 188–195. Springer, 1998.

[HB99] Hu and Blake. An improved diffusion algorithm for dynamic load balancing. *PARCOMP: Parallel Computing*, 25, 1999.

[HK00] Hendrickson and Kolda. Graph partitioning models for parallel computing. *PARCOMP: Parallel Computing*, 26, 2000.

[HL93] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993. appeared in Proc. Supercomputing 95.

[HL95] Bruce Hendrickson and Robert Leland. *The Chaco User's Guide Version 2*. Sandia National Laboratories, Albuquerque NM, 1995.

[HP-07] *HP-MPI User's Guide*, 11th edition, September 2007.

[KK98a] George Karypis and Vipin Kumar. *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*, September 1998.

[KK98b] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 10 January 1998.

[KK99] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[KL72] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Journal*, 49:291–307, 1972.

[MF01] Csaba Andras Moritz and Matthew I. Frank. LoGPC: Modeling network contention in message-passing programs. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):404–415, 2001.

[MYY+01] Sangman Moh, Chansu Yu, Hee Yong Youn, Ben Lee, and Dongsoo Han. Mapping strategies for switch-based cluster systems of irregular topology. In *ICPADS*, pages 733–740, 2001.

[Pel07] François Pellegrini. *SCOTCH 5.0 Users Guide*, August 2007.

[PR96] François Pellegrini and Jean Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical Report 1038-96, Université Bordeaux I, 1996.

[SOHL+98] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference. Volume 1, The MPI-1 Core*. MIT Press, Cambridge, MA, USA, 1998.

[The] The Top 500 Project. TOP500 Supercomputing Sites. http://www.top500.org.

[Trä02] Jesper Larsson Träff. Implementing the MPI Process Topology Mechanism. *SC Conference*, 2002.

[Trä06] Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

[Wal02] Chris Walshaw. *The serial JOSTLE library user guide: Version 3.0*. London, UK, July 2002.

[WC98] C. Walshaw and M. Cross. Mesh partitioning: a multilevel balancing and refinement algorithm. Tech. Rep. 98/IM/35, University of Greenwich, London SE18 6PF, UK, March 1998.

[WC01] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Computer Systems*, 17(5):601–623, March 2001.

[YCM06] Hao Yu, I-Hsin Chung, and José E. Moreira. Blue Gene System Software - Topology Mapping for Blue Gene/L Supercomputer. In *SC*. ACM Press, 2006.